UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Estudio e implementación de una técnica de clustering dinámico para trabajar con flujos de datos.

AUTORES: Molina, Roberto Pedro.

DIRECTOR: Hasperué, Waldo.

CODIRECTOR:

ASESOR PROFESIONAL:

CARRERA: Licenciatura en Informática.

Resumen

El objetivo general de esta tesina es estudiar y analizar las técnicas y problemáticas existentes de clustering (agrupamiento) aplicadas sobre los flujos de datos, buscando técnicas que permitan un agrupamiento dinámico. También, se realizará una investigación y estudio sobre los frameworks o plataformas de procesamiento de flujos de datos actuales con el fin de analizar la viabilidad para generar técnicas de clustering sobre estos entornos. Tras los resultados de las investigaciones y estudios previos, se propone como objetivo particular para esta tesina, el desarrollo, implementación, evaluación y comparación de un algoritmo de clustering dinámico aplicado al tratamiento de flujos de datos.

Palabras Clave

Flujos de datos, DataStream, Cluster analysis, Data Streaming Clustering, Streaming processing, Apache Spark Streaming, Minería de Datos, Machine Learning, aprendizaje no supervisado, programación distribuida, Two-phase Learning, Silhouette, Time Window.

Conclusiones

Se presenta D3CAS, un algoritmo de clustering dinámico basado en densidad para el procesamiento de flujos de datos, por lo que el objetivo de diseñar e implementar una técnica de clustering dinámica se pudo lograr satisfactoriamente, brindando además, una solución capaz de trabajar en un entorno distribuido y escalable gracias a que la implementación fue llevada sobre el motor de Spark Streaming.
Los resultados obtenidos en los experimentos y comparaciones con otros algoritmos de clustering son alentadores, logrando una buena calidad en todos los datesets analizados.

Trabajos Realizados

Estudio del modelo de flujo de datos. Estudio de diferentes metodologías de tratamiento de flujos de datos. Investigación sobre el estado del arte de técnicas de Streaming Clustering. Análisis de diferentes enfoques y metodologías de Streaming Clustering. Investigación sobre Apache Spark y Apache Spark Streaming. Instalación, configuración y pruebas de conceptos sobre Spark Streaming. Investigación de procesos de debugging sobre Spark. Implementación de simuladores de flujos de datos a partir de datasets para realizar benchmark sobre tareas de clustering. Diseño e implementación de una técnica de clustering dinámico sobre flujos de datos (D3CAS). Evaluación de la calidad de los resultados obtenidos por D3CAS. Comparación de resultados entre D3CAS y Clustream.

Trabajos Futuros

Como trabajo futuro se propone la ejecución de D3CAS en un ambiente distribuido que permitan medir y mejorar el rendimiento del algoritmo presentado, con el fin de realizar comparaciones tanto a nivel de resultados como también a nivel de consumo de recursos como memoria, procesamiento, overhead de comunicación, consumo energético, etc. Otro aspecto a estudiar es el de ejecutar pruebas en un entorno real, consumiendo un flujo de datos real, como por ejemplo, se podría consumir los flujos de datos que brindan los servicios de Twitter o de redes de sensores.

Fecha de la presentación: 07-20118

Resumen

El objetivo general de esta tesina es estudiar y analizar las técnicas y problemáticas existentes de clustering (agrupamiento) aplicadas sobre los flujos de datos, buscando técnicas que permitan un agrupamiento dinámico. También, se realizará una investigación y estudio sobre los frameworks o plataformas de procesamiento de flujos de datos actuales con el fin de analizar la viabilidad para generar técnicas de clustering sobre estos entornos.

Tras los resultados de las investigaciones y estudios previos, se propone como objetivo particular para esta tesina, el desarrollo e implementación de un algoritmo de clustering dinámico aplicado al tratamiento de flujos de datos, y al mismo tiempo, generando la documentación necesaria para su utilización.

A parte de esto, también se propone realizar una evaluación de los resultados obtenidos utilizando métricas especializadas para la evaluación de los resultados alcanzados, junto con una comparación de los resultados obtenidos con otros algoritmos existentes de clustering de flujos de datos. Los algoritmos serán ejecutados sobre flujos de datos simulados y creados con diferentes distribuciones conocidas para medir la capacidad de adaptación a los cambios dinámicos de las implementaciones realizadas.

Índice de Figuras

Índice de Tablas

Prefacio

En los últimos años, el progreso en las tecnologías de hardware ha hecho posible que las organizaciones y empresas puedan generar y almacenar grandes cantidades de datos. Cuando estos datos o conjuntos de datos se generan de manera continua y rápida en el tiempo siendo imposible de almacenarlos en su totalidad se los denominan '**Flujos de datos**', o en inglés Data Streams.

El procesamiento y análisis de un flujo de datos, datastream, es una característica que se encuentra presente en muchas aplicaciones y sistemas de software de la actualidad. Las transacciones simples de la vida cotidiana, como el uso de una tarjeta de crédito, un smartphone o un navegador web, han llevado a un almacenamiento automatizado de la información que se genera día tras día. En muchos casos, estos grandes volúmenes de datos pueden ser *minados*[1] para obtener información interesante y relevante en una amplia variedad de aplicaciones. De hecho, las aplicaciones de hoy en día que realizan un análisis, procesamiento o generan flujos datos se están haciendo más numerosas y más importantes, por ejemplo, aplicaciones de detección de intrusiones en la red, flujos de transacciones, registros telefónicos, redes sociales (y aplicaciones que se relacionan con estas), monitoreo en tiempo real de sensores, las tecnologías asociadas a Internet de las Cosas (IoT), avances tecnológicos en la medicina, monitoreo del clima, entre otras.

Además, en el campo de la Investigación, hay una rama muy activa sobre cómo almacenar, consultar, analizar, extraer y predecir la información relevante del flujo de datos analizado. La comunidad de la Minería de datos ha crecido rápidamente en los últimos años, y el tema de los flujos de datos es una de las áreas de interés más relevantes y actuales. Esto se debe al rápido avance del campo de los flujos de datos en los últimos años.

No obstante, el procesamiento de la información procedente del flujo de datos es uno de los problemas que en la actualidad pertenece al área de Big Data. Debido al crecimiento de Internet, la automatización de sistemas, el aumento de la conectividad social y el avance en la tecnología, las aplicaciones generan flujos de datos potencialmente infinitos, volátiles y continuos, lo que requiere un procesamiento en tiempo real, simple y rápido. A parte de esto, almacenar todos los datos del flujo es totalmente impráctico e ineficiente, lo que conlleva a

[1] *'Minados'*. Refiere al proceso de la minería de Datos que se encarga de extraer información de un conjunto de datos y transformarla en una estructura comprensible para su uso posterior.

pensar soluciones en un nuevo paradigma del tratamiento de la información donde cada dato se recibe, se usa una única vez y luego se descarta.

Por otra parte, el agrupamiento (clustering) es una tarea clave de la minería de datos y consiste en dividir o agrupar la información de tal manera que los datos en cada grupo (propiedad intra-cluster) sean similares y los datos entre grupos (propiedad inter-cluster) sean disímiles. Otro de los objetivos de la agrupación es reducir la complejidad de los datos mediante la sustitución de un grupo (cluster) de elementos/observaciones con una observación representativa, siendo este objetivo un desafío atractivo para atacar la problemática del gran volumen de datos dentro de un flujo. Los algoritmos de clustering suelen iterar sobre el conjunto de datos más de una vez, es por ello que las restricciones de tiempo de ejecución y uso de la memoria deben considerarse cuidadosamente en el contexto del análisis de los flujos de datos.

Una de las características más relevantes de los flujos de datos es que la distribución de éstos cambian continuamente. Por ello, es importante investigar sobre técnicas de clustering dinámico donde la cantidad de clusters en un momento dado dependerá de la distribución de los datos del flujo.

Con la creciente importancia de las aplicaciones de data stream mining, se han propuesto muchas plataformas para el tratamiento de estos. Pueden clasificarse en dos categorías: tradicionales o no distribuidos como MOA y plataformas distribuidas de streaming como Spark Streaming y Flink. Estas dos últimas, son consideradas como las plataformas de streaming más ampliamente utilizadas. Estos sistemas de flujo distribuido se basan en dos modelos de procesamiento, record-at-a-time (registro-a-tiempo) y micro-batching. En un modelo de procesamiento de record-at-a-time, los registros se procesan a medida que llegan, actualizan el estado interno y emiten nuevos registros. Por otro lado, el modelo de procesamiento de micro-batching ejecuta cada flujo como una serie de computaciones sobre lotes de datos en intervalos de tiempo pequeños, los cuales se implementan en Spark Streaming.

Dado todo este contexto, en esta tesina se pretende abordar un análisis tanto de las características de los flujos de datos como de las técnicas presente de clustering sobre flujos, a fin de desarrollar una nueva técnica de clustering con la ventaja de detectar dinámicamente la cantidad de cluster a formar. Es por este objetivo que la estructura del presente trabajo se encuentra ordenado en los siguientes capítulos:

En el capítulo 1, se lleva a cabo la presentación del modelo de Flujos de datos, explicando el origen de este modelo y su diferenciación de los sistema de almacenamientos tradiciones.

Junto a esto también se presentan las características y restricciones que tiene el modelo de Flujo de datos.
En el capítulo 2, tras haber investigado el motor Apache Spark, se presenta en este capítulo un resumen de las características y modo de procesamiento de este framework para el manejo de Big Data y DataStream, el cual viene siendo uno de los framework de mayor peso en la actualidad en este área.
Luego de entender los conceptos preliminares de los capítulos anteriores, en el capítulo 3 se describe los algoritmos más importantes dentro del estado del arte de las técnicas de clustering sobre flujos de datos destacando para cada uno de estos sus características, ventajas y desventajas.
En el capítulo 4, se presentan las características esenciales que debe tener un buen algoritmo de clustering y se presenta el diseño del algoritmo de clustering de esta tesina, D3CAS: “Distributed Dynamic Density based Clustering Algortihm for DataStream”, el cual como se indicó más arriba, presenta la característica de poder detectar dinámicamente agrupaciones sobre un flujo de datos.
En el capítulo 5, tras haber implementado el algoritmo D3CAS, se realizan las correspondientes evaluaciones para esta técnica haciendo hincapié en la detección dinámica de clusters y junto a esto también se comparan los resultados obtenidos contra uno de los algoritmos más populares dentro del área, Clustream. Para finalizar este trabajo, se presentan las conclusiones a las que se llegaron y las posibles ramas de investigación que se pueden continuar tras esta Tesina.

Capítulo 1

Flujos de datos

Introducción

En este capítulo, se presentará el modelo de Flujos de Datos, también llamado en inglés 'DataStreams', se hará una explicación de que son los Flujos de Datos, su diferenciación con los sistemas de bases de datos tradicionales, y las características principales que presentan como así también las restricciones que desencadenan.

Origen

Los sistemas de bases de datos tradicionales (DBMS) se utilizan en aplicaciones que requieren almacenamiento persistente de datos y consultas complejas. Por lo general, una base de datos consiste en un conjunto de objetos, con inserciones, actualizaciones y eliminaciones que ocurren con menos frecuencia que las consultas (lectura de la información), y donde las respuestas de las consultas reflejan el estado actual de la base de datos, este modelo representa adecuadamente información estática como catálogos comerciales o repositorios de información personal.

Sin embargo, en los últimos años hemos sido testigos del surgimiento de aplicaciones que no se ajustan a este modelo de datos y cuestionan el paradigma. En este nuevo enfoque, podemos reconocer una nueva clase de aplicaciones intensivas en datos: aplicaciones en las que los datos se modelan mejor no como relaciones persistentes, sino como flujos de datos **transitorios**[3]. Los ejemplos de tales aplicaciones incluyen aplicaciones financieras, monitoreo de red, seguridad, administración de datos de telecomunicaciones, redes sociales, aplicaciones web, fabricación, redes de sensores y otros.

En todas las aplicaciones citadas anteriormente, no es factible simplemente cargar los datos que llegan en un sistema de gestión de bases de datos (DBMS) tradicional y operar sobre este sistema. Los DBMS tradicionales no están diseñados para la carga rápida y continua de datos individuales, y no admiten directamente las consultas continuas que son típicas de las aplicaciones de flujo de datos.

Estos flujos o secuencias de datos son demasiado grandes para ser almacenados en la memoria principal y, alternativamente, se almacenan en dispositivos de almacenamiento secundarios. Por lo tanto, el acceso aleatorio a estos conjuntos de datos, que comúnmente se asume en la minería de datos tradicional, es altamente costoso[1] [2] [3]. Uno de los objetivos de la minería de flujo de datos es crear un proceso de aprendizaje que aumente linealmente el uso de recursos del sistema de acuerdo con la cantidad de ejemplos. Además, a medida que los datos llegan continuamente con nueva información, el modelo que se indujo previamente no solo necesita incorporar nueva información, sino que también elimina los efectos de los datos desactualizados. Simplemente volver a entrenar al modelo junto con los nuevos ejemplos más todos los ejemplos anteriores es ineficaz e inadecuado, por lo tanto, otro objetivo de la minería sobre flujo de datos es actualizar su modelo gradualmente a medida que llega cada ejemplo.

El Modelo de Flujos de datos

En el modelo de flujo de datos, los datos de entrada con los que se van a operar no están permanentemente disponibles para el acceso aleatorio desde la memoria o para recuperar desde un disco, sino que llegan como una o más secuencias continuas de datos temporales [2]. Los flujos de datos difieren del modelo de almacenamiento convencional de varias maneras:

1. Toda la información que forma un flujo de dato no está almacenada sino que se encuentra online y se actualiza de forma continua en el tiempo.
2. El sistema no tiene control sobre el orden en el que los elementos de datos llegan para ser procesados, ya que el orden depende enteramente de los factores y características que tiene el flujo de datos.
3. Los flujos de datos tienen un tamaño potencialmente ilimitado, lo que lleva a que estos sean procesado de manera 'on-the-fly', es decir, una vez recibido cierto elemento, este es procesado es ese mismo momento.
4. Una vez que se ha procesado un elemento de una secuencia de datos, se descarta o archiva. Los elementos ya procesados, no pueden ser recuperados fácilmente del flujo de dato (debido a la primera y segunda característica) a menos que se almacene explícitamente en la memoria, lo cual generalmente es imposible ya que es pequeña en relación con el tamaño de las secuencias de datos.
5. Operar en el modelo de flujo no excluye la presencia de algunos datos en bases de datos tradicionales. A menudo, las consultas de flujo de datos pueden realizar uniones entre flujos y datos relacionales almacenados.

Como conclusión, a partir de esta diferenciación, podemos definir a los flujo de datos o '*datastream'* como **una secuencia de elementos en tiempo real, continua, ordenada** (implícitamente por tiempo de llegada o explícitamente por timestamp), **variables en el tiempo y posiblemente impredecibles e ilimitados**, debido a que es imposible controlar en qué momento llegan los elementos, y tampoco es posible almacenar localmente una secuencia en su totalidad.

A partir de esto, podemos definir simple y formalmente a un flujo de datos como una secuencia:

$$DS = \{x_1, x_2, ..., x_i, ...\}$$

donde x_i es el objeto i-ésimo que llegó al flujo y en la mayoría de los flujos $x_i = (s, \delta)$ donde (s, δ) representa un par ordenado, siendo s un dato o un conjunto de datos y sea δ la forma de especificar el orden de s dentro del flujo, por ejemplo, δ podría ser un timestamp.

Características como Restricciones

Características	Minería de datos tradicional	Minería sobre flujos de datos
Números de pasadas (lecturas)	Múltiples	Única
Tiempo de ejecución	Ilimitado	Tiempo real
Restricción de Memoria	Ninguna	Limitada al tamaño de la RAM
Numero fuentes de datos (concept)	Única	Múltiples
Resultados	Precisos	Aproximado

Tabla 1.1: comparativa entre la minería tradicional y la minería sobre los flujos de datos.

La tabla anterior muestra la comparación entre la minería de datos tradicional y la minería sobre flujo de datos. La minería de datos tradicional puede escanear/leer conjuntos de datos muchas veces; se ejecuta generalmente sin restricciones de tiempo ni de memoria; tiene solo un concepto (fuente de datos); y necesita producir resultados bastante precisos[6]. Por otro lado, la minería sobre flujo de datos puede producir resultados aproximados y debe cumplir con ciertas restricciones, como ejecución en una única lectura (single-pass),

respuestas en tiempo real, memoria limitada, y detectar los cambios en la distribución de los datos (concept-drift):

- **Single-pass:** a diferencia de la minería de datos tradicional que puede leer datasets estáticos repetitivamente, cada muestra en un flujo de datos se examina a lo sumo una vez y no se puede volver a leer más adelante, por ende, no se puede rastrear (backtracking)[1]. La razón es que en muchos casos almacenar el gran volumen de información requiere usar un espacio de memoria secundario, y por ende las operaciones de E/S son bastante más caras (en tiempo de respuesta) que las operaciones de memoria principal. Esta restricción se puede relajar un poco de manera que un algoritmo tenga permiso solamente para recordar ejemplos en corto plazo. Por ejemplo, un algoritmo puede almacenar un lote de ejemplos con el objetivo de mantener una referencia histórica de los datos. Sin embargo, más tarde, debe descartar los datos almacenados, con el objetivo de prepararse para procesar los datos recién llegados.

- **Respuesta en tiempo real:** muchas aplicaciones de flujo de datos, como la predicción del mercado de valores, requieren una respuesta en tiempo real. La cantidad de tiempo para procesar los datos y proporcionar una decisión o resultado debe ser rápida, es decir, como los nuevos datos llegan constantemente incluso cuando se están procesando los datos anteriores; por lo que tiempo de cálculo por elemento también debe ser baja, de lo contrario la latencia del cálculo será demasiado alta y el algoritmo no podrá seguir el ritmo de la secuencia de datos, produciendo que se descarten elementos sin procesar[8][9].

- **Memoria limitada:** la cantidad de datos que llegan es extremadamente grande o potencialmente infinita. Como solo podemos computar y almacenar un pequeño ‘resumen’ de las secuencias de datos, la mayoría de los datos originales serán descartados, por lo tanto, los resultados aproximados son aceptables. Además, aunque existen algoritmos para manejar grandes volúmenes de datos sobre la memoria externa o secundaria, estos no son aceptables, ya que el uso de este tipo de memoria suelen ser demasiado lentos para las respuesta en tiempo real.

- **Detección de Concept-drift:** Concept-drift[7] se refiere a la situación donde los patrones descubiertos hasta el momento (o la distribución de dato actual) cambian con el tiempo. Esta propiedad no se presenta, en la minería tradicional, ya que al tener fuente de datos estáticas, la distribución de los elementos siempre es la misma, pues no se agregan nuevos datos.

Consultas sobre Flujos de datos

Las consultas sobre flujos de datos continuos tienen mucho en común con las consultas en un sistema de gestión de bases de datos tradicional. Sin embargo, hay dos distinciones importantes[2]:

La primera distinción es entre **consultas puntuales** (*one-time*) y **consultas continuas** [10]. Las consultas puntuales, las que se utilizan en los DBMS tradicionales, son consultas que se evalúan una vez en un instante puntual del conjunto de datos y devuelven directamente una respuesta al usuario. Las consultas continuas, por otro lado, se evalúan continuamente a medida que llegan datos al flujo. Las consultas continuas son la clase más interesante de consultas de flujo de datos, debido a que la respuesta a una consulta continua se produce a lo largo del tiempo, siempre reflejando los datos de la secuencia vistos hasta el momento. Las respuestas de consultas continuas pueden almacenarse y actualizarse a medida que llegan nuevos datos o pueden producirse como flujos de datos.

La segunda distinción es entre **consultas predefinidas** y **consultas ad hoc**. Una consulta predefinida es aquella que se suministra al sistema de gestión de flujo de datos antes de que haya llegado cualquier dato relevante. Las consultas predefinidas generalmente son consultas continuas, aunque las consultas one-time también pueden ser predefinidas. Las consultas ad hoc, por otro lado, se emiten en línea una vez que las transmisiones de datos ya han comenzado. Las consultas ad hoc pueden ser consultas únicas o continuas. Las consultas ad hoc complican el diseño de un sistema de gestión de flujo de datos, tanto porque no se conocen de antemano con fines de optimización de consultas, identificación de subexpresiones comunes entre consultas, etc., y más importante aún, porque la respuesta correcta a una consulta ad hoc puede requerir referenciar elementos de datos que ya han llegado a las secuencias de datos y que posiblemente ya se hayan descartado.

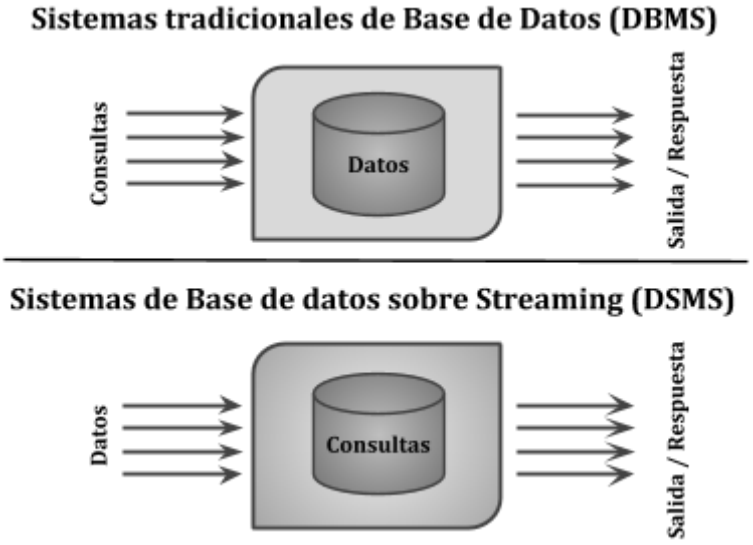


Figura 1.1: Diferenciación entre sistema de base de datos

Como se ve en la Figura 1.1, se ejemplifica gráficamente las diferencias entre las consultas continuas predefinidas manejadas en un sistema de flujo de datos y las consultas on-time ad-hoc manejadas por un DBMS, en la imagen se puede apreciar que en el caso de un DBMS cada vez que llega una consulta, se procesa y se genera una única respuesta con los datos almacenado hasta el momento. Por otra lado, en el caso de los sistemas de flujos de datos, las consultas ya se encuentra cargadas y se va actualizando la respuesta a medida que llegan las secuencia de datos.

Modelo general para un algoritmo de Data Streaming

Para comprender una visión general de la minería sobre flujos de datos, presentamos en la figura 1.2, un modelo general para el procesamiento de flujos de datos [6]: Cuando llega una secuencia, se utiliza un almacenamiento intermedio (búfer) para almacenar los elementos más recientes. El motor de streaming lee el búfer para crear un *resumen o sinopsis* y lo guardar en la memoria principal. Para mantener esta información actualizada, el sistema puede aplicar diferentes enfoques o métricas como por ejemplo, métricas de tiempo (time window).

Cuando ocurre algún determinado evento, por ejemplo, la solicitud de un usuario o el paso de un cierto lapso de tiempo; el motor de streaming procesa el resumen (sinopsis) y obtendrá algunos resultados aproximados. En general, la mayoría de los algoritmos de flujo de datos son adaptaciones de los algoritmos de minería tradicionales. Opcionalmente, se puede utilizar una etapa de pruebas/testing, donde es posible hacer una validación del modelo generado.

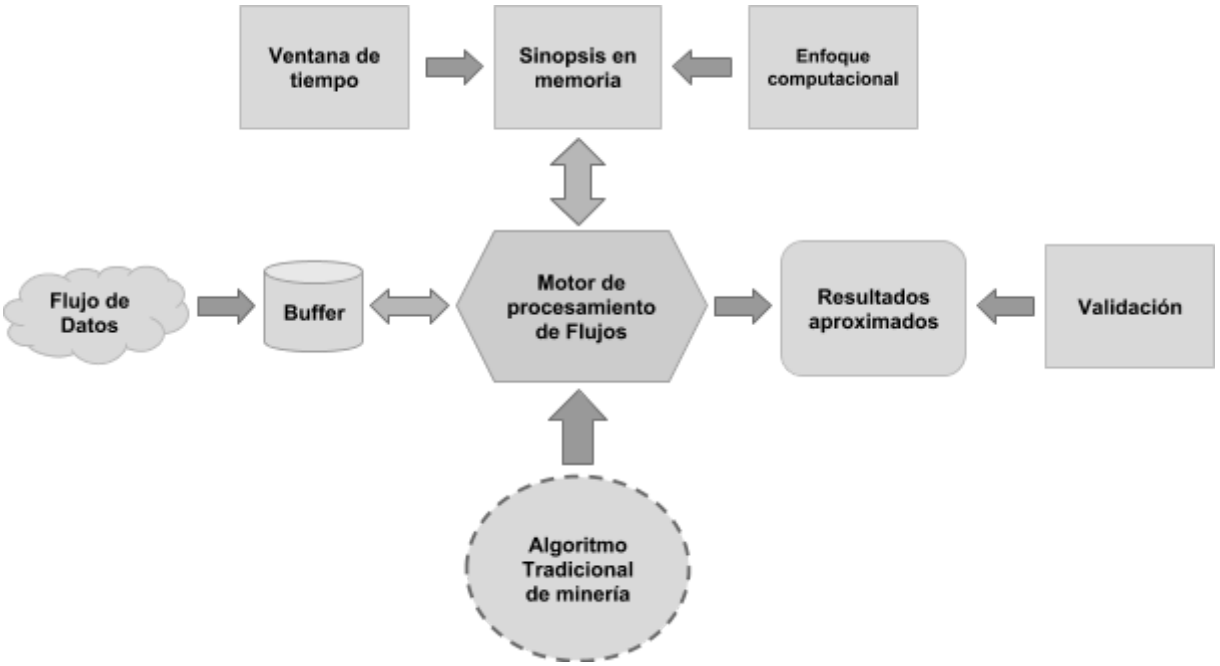


Figura 1.2: gráfico general del procesamiento de Flujos de datos.

Ventanas de tiempo

Como los flujos son potencialmente infinitos, sólo es posible procesar una parte de su totalidad. A la porción de datos sobre la cual vamos a trabajar se la define como una ventana temporal de objetos:

$$W[i,j] = (x_i, x_{i+1}, x_{i+2}, \ldots, x_j)$$

donde i y j son puntos en el tiempo y también se cumple $i < j$.

Hay diferentes tipos de ventanas de tiempo: landmark window, sliding window, fading window, y tilted time window.

Landmark window: En este tipo de ventanas, estamos interesados en todos los elementos de la secuencia a partir del instante de tiempo 0 (es decir, desde que comienza la transmisión de datos) hasta la hora actual (t_c). Esta ventana se representa como $W[0, t_c]$ [11][13].

Cuando usamos este tipo de ventanas, todas las transacciones son igual de importantes; no hay diferencia entre los datos pasados y los presentes. Sin embargo, a medida que el flujo continuamente evoluciona, el modelo creado a partir de objetos antiguos puede volverse inconsistente con los objetos nuevos, debido al cambio de distribución de los datos a medida que pasa el tiempo. Si se tiene como objetivo considerar o hacer hincapié en los datos más

recientes, se deben utilizar otros tipos de ventanas como sliding window, tilted window, o fading window.

Figura 1.3: Ejemplo de Landmark Window

Sliding window: Esta variante se representa como $W[t_{c-w+1}, t_c]$, donde solamente nos interesan las w transacciones más recientes, las demás son descartadas. Por lo tanto, los resultados dependen del tamaño de la ventana w [13] . Si w es demasiado grande y existe *concept drift*, la ventana posiblemente contenga información obsoleta o inconsistente, por lo que la precisión del modelo disminuye. Si w es pequeño, la ventana puede tener poca información (déficit de elementos) y el modelo sufre de *overfitting*[2] (sobreajuste) y por ende, sufre de variaciones importante es los resultados a lo largo del tiempo.
Recientemente, se han propuesto ventanas deslizantes flexibles donde el tamaño de la ventana cambia de acuerdo con la precisión del modelo. Cuando la precisión es alta, la ventana se extiende; y cuando la precisión es baja, la ventana se encoge.

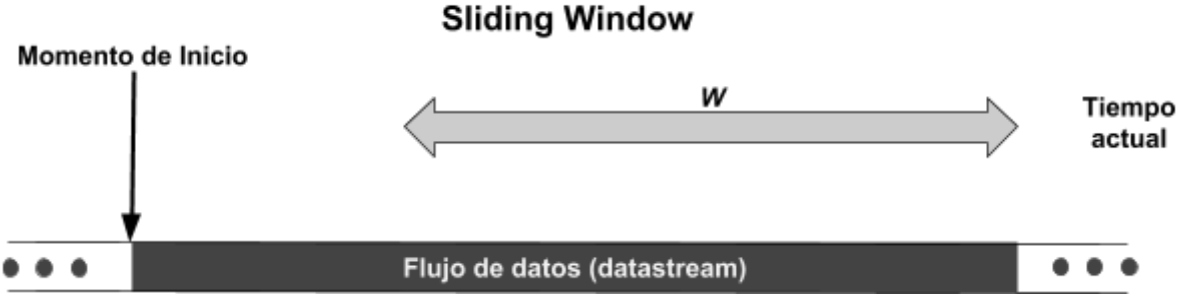


Figura 1.4: Ejemplo de Sliding Window con tamaño w.

Fading window: también conocida como Damped window**,** En esta representación, a cada objeto de datos se le asigna un peso diferente de acuerdo con su tiempo de llegada, donde el peso es inversamente proporcional a la edad del objeto, a fin que las nuevas

[2] Overfitting: efecto del entrenamiento de un algoritmo de aprendizaje automático, en el cual el modelo se ajusta muy bien a los datos existentes pero tiene un pobre rendimiento para predecir nuevos resultados.

transacciones reciban pesos mayores que los antiguos [14]. Usando fadding window (ventana de desvanecimiento), reducimos el efecto (importancia) de las transacciones antiguas y obsoletas sobre los resultados. Generalmente se utiliza una función exponencial decreciente

$$f(\Delta t) = \lambda^{\Delta t} (0 < \lambda < 1)$$

en este modelo de ventana. Donde Δt es la edad (age) de un objeto, lo cual se representa como la diferencia de tiempo entre la hora actual y su hora de llegada. Este modelo, necesita elegir un parámetro de desvanecimiento λ adecuado, que generalmente se establece en el rango [0, 1] en aplicaciones reales y un valor límite, donde los elementos con pesos menores a este limite seran descartados.

La diferencia entre sliding window y fading window, es que en el modelo de sliding tenemos una cantidad w fija de elementos en una ventana, mientra que en el modelo fading tenemos una cantidad variable y no predecible de elementos [2], pues el peso de cada elemento dependen de la velocidad con la que llegan los datos y pasan por la función de fading, marcando una relación directamente proporcional entre la cantidad de elementos junto a la velocidad del stream. esto quiere decir, que a mayor velocidad de llegada de elemento, mayor van ser los elementos que se encuentran en la ventana.

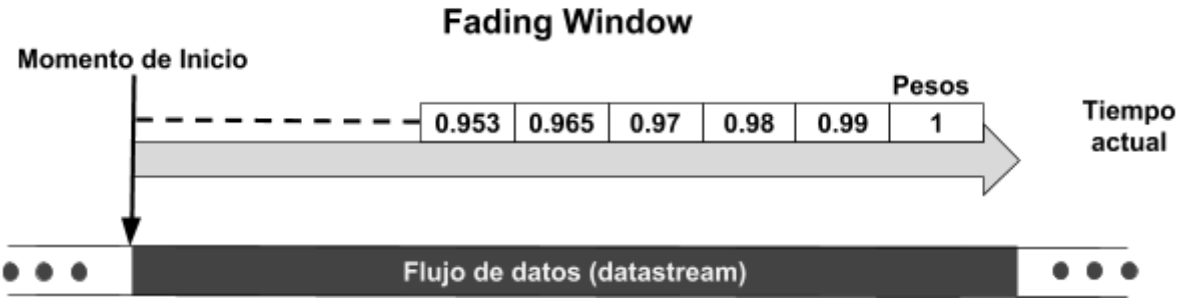


Figura 1.5: Ejemplo de Fading Window.

Tilted time window: este modelo es una variante que se encuentra entre el modelo de sliding window y fading window[5]. Aplica diferentes niveles de granularidad con respecto a la actualidad de los datos. Cuando los datos son recientes se tiene una granularidad en escala fina, es decir, mayor precisión, y cuando los datos son menos reciente se tiene una granularidad en una escala gruesa. Este modelo almacena aproximadamente todo el conjunto de datos y proporciona una buena compensación entre los requisitos de almacenamiento y la precisión. Sin embargo, el modelo puede volverse inestable después de ejecutarse durante un tiempo prolongado. Por ejemplo, la estructura de árbol en

FP-Stream[27] será muy grande con el tiempo, y el proceso de actualización y escaneo sobre el árbol puede degradar su rendimiento.

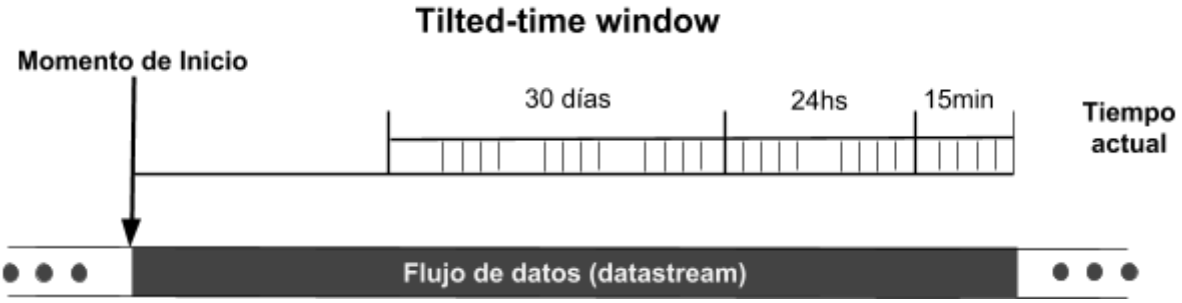


Figura 1.6: Ejemplo de Titled-time Window.

Enfoques Computacionales

Mientras que al utilizar las ventanas de tiempo descriptas anteriormente, decidimos a qué porción del flujo vamos a analizar o a qué información le vamos a dar prioridad, también hay que pensar cómo se va a procesar los datos sobre estas ventanas, es por esto que existen dos enfoques computacionales para procesar los flujos de datos.

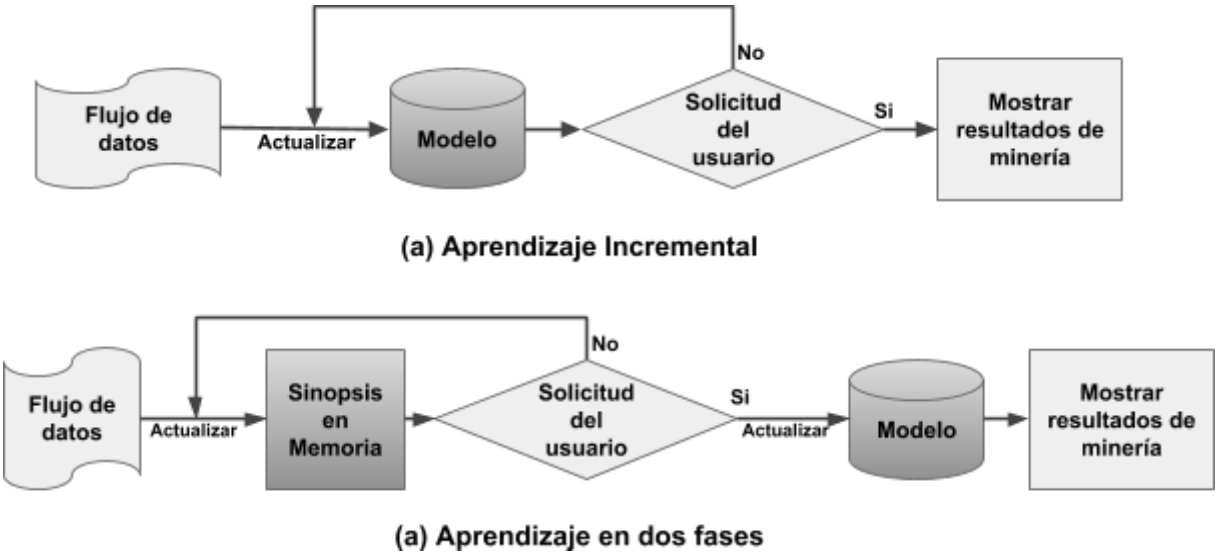


Figura 1.7: Diferencia entre el enfoque incremental y el enfoque de dos fases

Aprendizaje incremental (Incremental learning): En este enfoque, el modelo evoluciona incrementalmente (progresivamente) para adaptarse a los cambios que presentan los datos entrantes en el flujo [15], [12]. Específicamente, el proceso de minería se actualiza cada vez que llegan elementos nuevos, es decir, que se actualizan los resultados o modelos tras procesar cada elemento entrante. Hay dos esquemas para actualizar el modelo: por

instancia de datos y por ventana. Este enfoque tiene la ventaja de proporcionar resultados de manera instantánea, pero requiere más recursos computacionales.

Por ejemplo, en el trabajo “A streaming ensemble algorithm (sea) for large-scale classification” [16] se implementó un conjunto de clasificadores para flujo de datos, sobre los cuales evalúa una ventana de datos entrantes y adapta el modelo ajustando el peso de cada clasificador o reemplazando un clasificador antiguo por uno actualizado.

Aprendizaje en dos fases (Two-phase Learning): también conocido como online-offline learning, es un enfoque computacional para el procesamiento de flujos de datos [5]. La idea básica es dividir el proceso de minería en dos fases o etapas. En la primera fase (etapa online), se va generando una sinopsis (*synopsis*) o resumen de los datos que llegan en tiempo real. En la segunda fase (etapa offline), el proceso de minería se realiza utilizando las sinopsis generadas y almacenadas en la etapa online, permitiendo procesar las sinopsis mientras llegan nuevos datos en la etapa online. Este enfoque puede procesar flujos de datos a alta velocidad. Sin embargo, su limitación es que los usuarios deben esperar hasta que los resultados de la etapa offline estén disponibles.

Por ejemplo, Aggarwal, et al. [5] propusieron un método de agrupamiento bajo este enfoque. Su componente online resume la información estadística de la secuencia de datos en tiempo real. Mientras tanto, el componente offline usa los resúmenes estadísticos generados para realizar la agrupación cuando sea necesario.

Aplicaciones

El procesamiento de flujos de datos, a pesar de ser una actividad que se ha empezado a realizar no hace mucho tiempo, ya cuenta con muchos tipos de usos o aplicaciones que tratan de explotar sus características para obtener información valiosa la cuales brindan ciertas facilidades o automatizar tareas según en el campo donde se estén aplicando. A continuación se presentan varias aplicaciones para los flujo de datos y los requisitos que se necesitan en cada caso para realizar análisis de sobre los datos.

Redes de sensores (Sensor networks): este tipo de redes, consta de sensores autónomos distribuidos espacialmente que supervisan de forma cooperativa un determinado entorno. Estos sensores pueden detectar los valores físicos del entorno, como la temperatura, el sonido, la vibración, la presión, la humedad y la luz. Pueden pasar sus datos de forma cooperativa a través de la red a un centro o sistema de monitoreo, el cual o bien puede procesarlos para una tarea determinada o prepararlos para ser enviados para su posterior procesamiento. Las redes de sensores están involucradas en muchas aplicaciones de la

vida real, como el monitoreo del tráfico, hogares inteligentes, monitoreo de hábitats y cuidado de la salud [17].

Por ejemplo, existen hospitales modernos que están equipados con un sistema de monitoreo de pacientes para mejorar la calidad de la atención médica y la productividad del personal. Muchos sensores corporales, que están conectados a pacientes críticamente enfermos, pueden producir datos fisiológicos masivos, como temperatura, electrocardiograma, oximetría de pulso y presión arterial. Como los sensores sólo almacenan datos actualizados y los ojos humanos no pueden detectar estas señales, el sistema debe analizar los flujos de datos en tiempo real y extraer información significativa para los médicos profesionales, como reglas clínicas para identificar distintas patologías o síndromes [18].

Flujos en redes sociales: las redes sociales en línea (OSNs: Online social networks) se han vuelto cada vez más populares; por ejemplo, Facebook, Twitter y LinkedIn tienen millones de usuarios activos. Dichas redes generan enormes flujos de datos en línea, como texto, multimedia, enlaces e interacciones. Hay mucha investigación sobre minería de redes sociales [20][21][22][23].

Por ejemplo, los métodos de agrupación de flujo se utilizan para detectar comunidades y monitorear su evolución en las redes sociales. Pueden explicar cómo emergen, se expanden, se contraen y evolucionan las comunidades en una red social [19]. Además, los algoritmos de clasificación de flujo ayudan a clasificar diferentes tipos de usuarios, categorizar temas de discusión o detectar eventos.

Mining query streams: buscar en la web para recuperar información se ha convertido en una actividad esencial de nuestra vida cotidiana. Los motores de búsqueda existentes, como Google, Bing y Yahoo, manejan millones de consultas a diario. La minería sobre los flujos de datos de estos motores ha atraído mucho trabajo de investigación y es un desafío interesante que puede proporcionar mejores resultados de búsqueda a los usuarios.

Por ejemplo, en “Learning to cluster web search results” [17] ha investigado cómo agrupar los resultados de búsqueda por medio de los patrones de consultas que realizan los usuarios, con el objetivo de brindarle a los usuarios una forma de navegación más rápida sobre estos.

Monitoreo de red (Network monitoring): Internet incluye muchos routers que están conectados y se comunican entre sí mediante el envío de paquetes IP. Para administrar dichas redes, necesitamos analizar los datos de tráfico para descubrir patrones de uso y actividades inusuales en tiempo real. Los datos de tráfico se registran en forma de archivos de registro, ejemplos, los registros de paquetes que contienen las direcciones IP de origen y

destino; logs de paquetes enviados, junto con la hora de inicio, la hora de finalización y el protocolo utilizado. Un ejemplo de administración de red es detectar y prevenir ataques maliciosos en una gran red de proveedores de servicios de Internet. Se requiere un clasificador de flujo de datos para clasificar en tiempo real diferentes tipos de ataques, como la denegación de servicio (DOS), el acceso no autorizado desde una máquina remota (R2L), el acceso no autorizado a los privilegios locales de superusuario (U2R), la vigilancia y otros ataques de sondeo [24][25][26].

Capítulo 2

Apache Spark

Introducción

Este capítulo, se centra en realizar un estudio y análisis del framework Apache Spark, el cual viene siendo uno de los framework de mayor peso en la actualidad en el área de Big Data tanto en la industria del software como en ambientes académicos, por lo que el objetivo de este capítulo es entender y brindar un resumen de sus principales características y modelo de procesamiento tanto para el manejo de Big Data en general como para procesamiento de FLujos de datos (DataStream). Por medio de este capítulo, el lector tendrá los conceptos básicos para comprender las decisiones tomadas en el diseño del algoritmo de esta tesina.

¿Qué es Apache Spark?

Apache Spark [33] es un sistema de cómputo distribuido de alto rendimiento y de propósito general diseñado para realizar procesamiento sobre un cluster de PCs. Proporciona por medio de su API, métodos que son generalizables para procesar datos en paralelo; las mismas funciones de alto nivel se pueden usar para ejecutar tareas sobre datos de diferentes tamaños y estructuras. Se ha convertido en el proyecto de código abierto de Apache más activo, con más de 1.000 colaboradores.

Spark extiende el modelo popular de MapReduce[28], adaptando su diseño y API con las características del framework distribuido DryadLINQ[29], con el objetivo de admitir y permitir de manera eficiente operaciones que generalmente no se realizan bajo este modelo, como las consultas interactivas (DBMS) y el procesamiento de flujos de datos (streams). Una de las características principales de Spark es la velocidad de ejecución, debido a que tiene la capacidad de ejecutar operaciones sobre los datos en la memoria principal que a diferencia del modelo tradicional de MapReduce se realizan en disco. Otra de las características más

importante de Spark es que tiene la capacidad de realizar evaluaciones lazy sobre las operaciones de memoria.

Spark es diseñado para ser altamente accesible, ofreciendo una API simple para diferentes lenguajes, entre ellos, Scala, Java, Python y SQL. También se puede integrar con otras herramientas de Big Data. En particular, Spark puede ejecutarse en clústeres de Hadoop y acceder a cualquier data source de Hadoop, como Cassandra.

El proyecto Spark contiene múltiples componentes integrados. En esencia, Spark es un "motor computacional" que se encarga de programar, distribuir y monitorear aplicaciones que consisten en muchas tareas sobre muchas máquinas (workers), en otras palabras, directamente sobre un cluster. Debido a que el core central de Spark es rápido y de uso general, permite utilizar múltiples componentes de alto nivel especializados para diversos tipos de trabajo, como SQL o aprendizaje automático. Estos componentes están diseñados para interoperar conjuntamente, permitiéndole al usuario combinarlos como bibliotecas dentro de sus proyectos.

La filosofía de integración de componentes en Spark tiene varios beneficios. En primer lugar, todas las bibliotecas y componentes de mayor nivel en la pila (stack) de Spark se benefician de las mejoras en las capas inferiores. Por ejemplo, cuando el motor central de Spark agrega una optimización, las bibliotecas SQL y de aprendizaje automático también se aceleran automáticamente. En segundo lugar, los costos asociados con la ejecución de la pila (stack) se minimizan, porque en lugar de ejecutar 5-10 sistemas de software independientes, con Spark solo se necesita ejecutar una. Estos costos incluyen implementación, mantenimiento, pruebas, soporte, integración y otros. Esto también significa que cada vez que se agrega un nuevo componente a la pila de Spark, todas las organizaciones que usan Spark podrán probar inmediatamente este nuevo componente. Esto cambia el costo de probar un nuevo software para el análisis de datos desde la descarga, implementación, actualización y aprendizaje.

Finalmente, una de las mayores ventajas de la integración propuesta por Spark, es la capacidad de crear aplicaciones que combinen diferentes modelos de procesamiento sin problemas. Por ejemplo, en Spark se puede escribir una aplicación de aprendizaje automático para clasificar los datos en tiempo real, ya que se ingieren o consumen a partir de un flujo de dato (streaming source). Simultáneamente, se pueden consultar los datos resultantes en tiempo real, a través de SQL (por ejemplo, para unir los datos con archivos de registro no estructurados). Además, los usuarios pueden acceder a los mismos datos a través del shell de Python para realizar un análisis ad hoc. Y otros pueden acceder a los datos utilizando aplicaciones batch. Mientras, que el equipo de IT sólo tiene que mantener solamente el sistema Spark.

Componentes de Spark

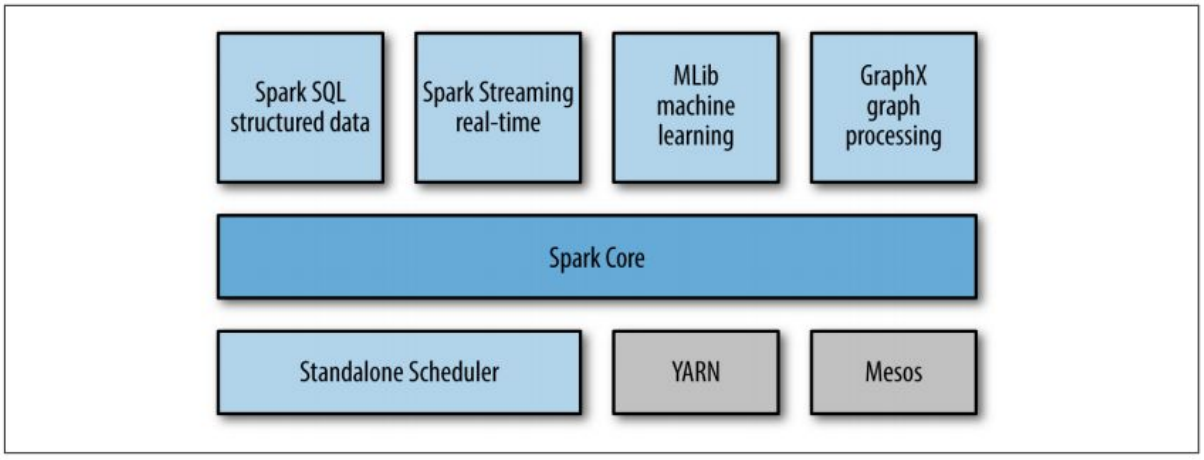


Figura 2.1: Pila (stack) de componentes de Spark

Spark Core

Spark Core contiene la funcionalidad básica de Spark, que incluye componentes para la programación de tareas (scheduling), administración de memoria, recuperación de fallas, interacción con sistemas de almacenamiento y otras funcionalidades. Spark Core también es el hogar de la API que define 'Resilient distributed datasets' (RDDs) (los conjuntos de datos distribuidos), que son la abstracción de programación principal de Spark. Los RDD representan una colección de elementos distribuidos en muchos nodos de cómputo que pueden manipularse en paralelo. Spark Core proporciona muchas API para diferentes lenguajes para compilar y manipular estas colecciones.

Spark SQL

Spark SQL es el paquete de Spark para trabajar con datos estructurados. Permite consultar datos a través de SQL, así como la variante Apache Hive de SQL, llamada Hive Query Language (HQL), y admite muchas fuentes de datos, incluidas tablas Hive, Parquet y JSON. Más allá de proporcionar una interfaz SQL para Spark, Spark SQL permite a los desarrolladores mezclar consultas SQL con las manipulaciones de datos programáticos compatibles con RDD en Python, Java y Scala, todo dentro de una sola aplicación, logrando la combinación de SQL con procesamientos y análisis complejos.

Spark Streaming

Spark Streaming es un componente Spark que permite el procesamiento de transmisiones o flujos de datos en vivo. Los ejemplos de flujos de datos incluyen desde archivos de registro generados por servidores web de producción a colas de mensajes que contienen actualizaciones de estado publicadas por los usuarios de un servicio web.

Spark Streaming proporciona una API para manipular flujos de datos utilizando la misma API de RDD del Spark Core, facilitando a los programadores aprender un único software capaz de realizar aplicaciones que manipulan datos almacenados en la memoria, en el disco o en tiempo real. Debajo de su API, Spark Streaming fue diseñado para proporcionar el mismo grado de tolerancia a fallas, rendimiento y escalabilidad que Spark Core. Especialmente en esta tesina, se estará utilizando este componente.

MLlib

Spark viene con una biblioteca que contiene funcionalidades para realizar tareas de Machine Learning (ML), llamada MLlib. MLlib proporciona múltiples tipos de algoritmos de aprendizaje automático, incluyendo clasificación, regresión, clustering y filtrado colaborativo, así como funcionalidades de soporte tales como evaluación de modelos e importación de datos. También proporciona algunas primitivas ML de nivel inferior, incluido un algoritmo genérico de optimización de descenso de gradiente. Todos estos métodos están diseñados para escalar a través de un clúster.

Cluster Managers

Spark está diseñado para escalar de manera eficiente de uno a muchos miles de nodos de computación. Para lograr esto mientras se maximiza la flexibilidad, Spark puede ejecutarse en una variedad de administradores de clusters, incluyendo Hadoop YARN, Apache Mesos, y además, Spark tiene su propio administrador de clúster, llamado Standalone Scheduler.

Arquitectura

Antes de profundizar en los detalles del modelo de Spark, es útil comprender la arquitectura distribuida de Spark. Spark utiliza una arquitectura maestro/esclavo (master/slave) con un coordinador central y muchos trabajadores distribuidos. El coordinador central se llama driver, master o controlador. El controlador se comunica con un número potencialmente grande de trabajadores distribuidos llamados executors o ejecutores. El controlador se ejecuta en su propio proceso de Java y cada ejecutor es un proceso de Java separado. Un controlador y sus ejecutores se denominan conjuntamente una aplicación Spark.

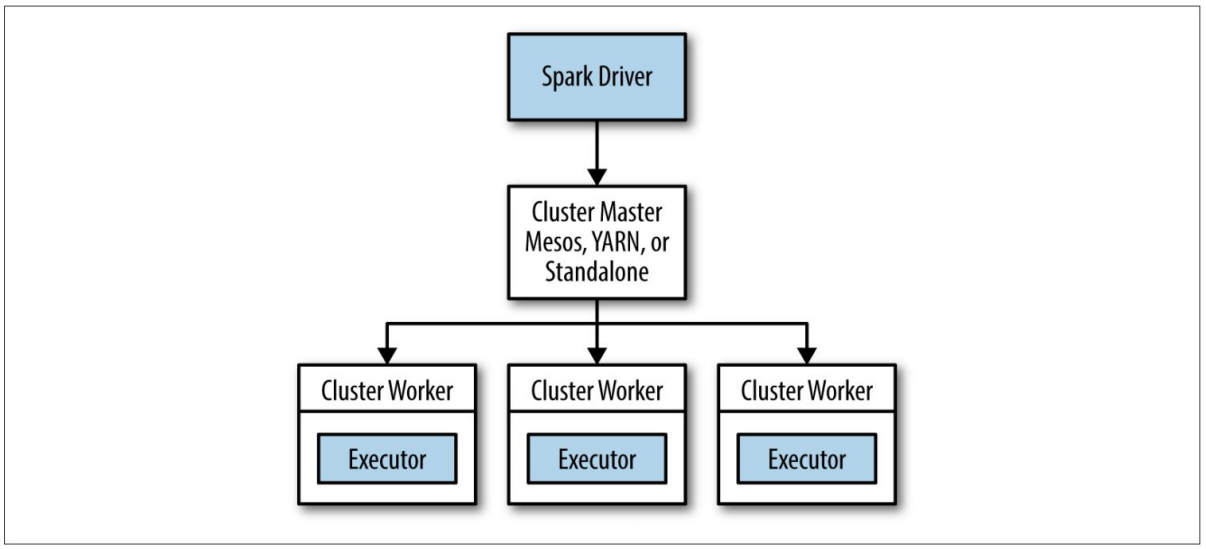


Figura 2.2: Componentes para la ejecución distribuida en Spark. Figura extraída de [30]

Modelo de procesamiento en paralelo: RDDs

Cada aplicación Spark consiste en un programa driver (o controlador) que inicia varias operaciones paralelas en un clúster. El programa driver contiene la función principal de la aplicación Spark creada por el usuario, en la cual se definen las fuentes de datos a utilizar que serán distribuidos en el clúster para luego realizar sobre éstas, operaciones en paralelo.

Spark representa las fuentes de datos con una estructura denominada RDDs (Resilient distributed datasets)[34] que son colecciones de objetos distribuidos e inmutables que se almacenan en los nodos executors (nodos ejecutores o nodos slaves). Los objetos que forman un RDD se denominan partitions (o particiones) y pueden (pero no necesitan ser) computados en diferentes nodos executores dentro del sistema distribuido. Los RDD pueden contener cualquier tipo de objetos Scala, Java o Python incluidos dentro de las clases definidas por el usuario.

Cuando el driver empieza su ejecución realiza las siguientes dos tareas:

- **Convertir el programa definido por el usuario en Tareas:** El driver es responsable de convertir un programa escrito por un usuario en unidades de ejecución física llamadas tareas. A un alto nivel de abstracción, todos los programas Spark siguen la misma estructura: crean RDD a partir de alguna entrada, derivan nuevos RDD por medio de transformaciones y realizan acciones para recopilar o guardar datos. Spark crea implícitamente un grafo acíclico dirigido (DAG) de operaciones. Cuando el driver se ejecuta, convierte este grafo en un plan de ejecución de múltiples tareas. Las tareas se agrupan y preparan para enviarse al clúster. Las tareas son la unidad de

trabajo más pequeña en Spark; un programa de usuario típico puede iniciar cientos o miles de tareas individuales.

- **Scheduling de tareas en los nodos ejecutores:** Cuando se inicia la aplicación de Spark, inmediatamente se inicia la ejecución del Spark cluster manager (o administrador de cluster de Spark), quien es el encargado de manejar el inicio y la distribución de los nodos ejecutores en un sistema distribuido de acuerdo con los parámetros de configuración establecidos por la aplicación. Luego de esto, el programa driver de Spark distribuye las particiones de los RDDs y por último, empieza a coordinar la distribución de tareas individuales sobre los ejecutores. Por lo tanto, cada ejecutor representa un proceso capaz de ejecutar tareas y almacenar particiones del RDD.

Spark puede mantener un RDD cargado en la memoria de los nodos ejecutores durante toda la vida útil de la aplicación con el objetivo de tener un acceso más rápido en cálculos repetidos. Otra característica importante de los RDD es que son inmutables, por lo que la transformación de un RDD devuelve un nuevo RDD en lugar de devolver el RDD actual pero con las modificaciones generadas por la transformación.

Funciones sobre RDD: Transformaciones y Acciones

Las operaciones de transformaciones y acciones son características heredadas o tomadas del motor de procesamiento DryadLINQ y constituyen unas de las características más importante en el diseño del modelo de Spark, debido a que por medio de estas, Spark brinda una interfaz de programación sencilla de alto nivel. Mediante la cual los usuarios pueden escribir programas secuenciales utilizando operaciones imperativas o declarativas sobre los datos, delegando automática y transparentemente al motor de Spark la tarea de transformar, optimizar y compilar dichos programas en tareas distribuidas con una eficiente paralelización de datos sobre un cluster de máquinas. Específicamente, las tareas que son generadas están diseñadas para trabajar bajo el modelo de MapReduce.

Los RDD admiten dos tipos de operaciones:

- Transformaciones, que crean un nuevo conjunto de datos a partir de uno existente, específicamente, crean un nuevo RDD a partir de otro RDD.

- Acciones, que devuelven un resultado final al programa controlador después de ejecutar un cálculo sobre los datos del RDD, pero el resultado de las acciones

siempre es algo distinto que un RDD, ya sea un valor, objeto, o una lista de un tipo de dato según el lenguaje que se esté utilizando.

Por ejemplo, la transformación map() que pasa cada dato dentro del RDD por una función y devuelve un nuevo RDD que representa los resultados de la función para cada dato. Por otro lado, reduce() es una acción que junta y procesa todos los elementos del RDD según un determinado criterio y devuelve el resultado final al programa driver.

Cada programa Spark debe contener una acción, ya que las acciones devuelven información al controlador, las acciones son lo que fuerzan la evaluación de un programa Spark, por lo tanto, siempre son las encargadas de realizar el procesamiento final de un RDD. Pero no es necesario que siempre devuelvan un resultado, también pueden desencadenar una función de tipo side effect, es decir, puede utilizar los datos de un RDD para cambiar el estado interno de la aplicación Spark, o escribir algún resultado en una base de datos, o enviar información a un endpoint de otra aplicación.

A continuación se presentan dos tablas (tabla 2.1, tabla 2.2) con las transformaciones y acciones más utilizadas definidas dentro de la API Spark:

Transformaciones definidas en la API

Transformación	Funcionalidad
map(*funcion*)	Genera un nuevo RDD a partir del resultado de aplicar ***funcion*** a cada elemento del RDD original.
filter(*funcion*)	Genera un nuevo RDD a partir de los elementos del RDD original que cumplen con la condición ***funcion***
flapMap(*funcion*)	Similar a ***map***, pero ***funcion*** en vez de devolver un único resultado puede devolver 0 o más elementos, por lo tanto, ***funcion*** debe devolver una secuencia de elementos. Luego, ***flapMap*** devuelve el resultado de concatenar todo los resultados producidos por ***funcion***.
distinct()	Genera un RDD a partir de eliminar los elementos duplicados del RDD original.
reduceByKey(*funcion*)	A partir de un RDD con pares de elementos (K, V), se crea un nuevo RDD con pares de elementos (K, V) donde los valores de cada clave K se procesan utilizando la función de reducción ***funcion***, que debe ser del tipo (V, V) => V.

union(*rdd*)	Genera un nuevo RDD a partir de los datos que se encuentran en la unión entre el RDD original y ***rdd***.
intersection(*rdd*)	Genera un nuevo RDD a partir de los datos que se encuentran en la intersección entre el RDD original y ***rdd***.
cartesian (rdd)	Genera un nuevo RDD a partir del producto cartesiano entre el RDD original y ***rdd***.

Tabla 2.1: Transformaciones Apache Spark.

Acciones definidas en la API

Acciones	Funcionalidad
collect()	Devuelve al programa driver un arreglo con todos los elementos del RDD.
count()	Devuelve la cantidad elementos que contiene el RDD.
countByKey()	Disponible sólo en los RDD de tipo (K, V). Devuelve un hashmap del tipo (K, int) que representa la cantidad de ocurrencia de la clave K.
take(*n*)	Devuelve un arreglo con los primeros ***n*** elementos del RDD.
reduce(*funcion*)	Acumula/combina todos los elementos del RDD utilizando ***funcion*** la cual se ejecuta en paralelo. ***funcion*** toma dos argumentos y devuelve un único resultado.
foreach (*funcion*)	Aplica ***funcion*** sobre cada elemento del RDD. **foreach** no devuelve un resultado. Se utiliza para generar side effects, como por ejemplo, modificar variables del programa o interactuar con un almacenamiento externo.
savaAsText(*path*)	Escribe todos los elementos del RDD en un archivo de texto dentro del directorio local al driver definido por ***path***. Spark ejecutará toString para cada elemento del RDD y se corresponderá con cada línea dentro del archivo.

Tabla 2.2: Acciones Apache Spark

Evaluación lazy

En lugar de evaluar cada transformación tan pronto como sea posible según como se especifica en el programa del controlador, Spark evalúa los RDD de forma perezosa (lazy evaluation), es decir, calculando las transformaciones de RDDs solo cuando los datos resultantes realmente sean requeridos, es decir, cuando se necesite realizar una operación

de acción. En simples palabras, Spark no comenzará a ejecutar ninguna transformación sobre las particiones de un RDD hasta que no se invoque una acción.

Una acción es una operación Spark que devuelve un formato distinto de un RDD, lo que desencadena la evaluación del RDD y posiblemente genere una salida (output) para un sistema fuera del modelo de Spark; por ejemplo, devolver los datos al driver (por medio de las operaciones *count* o *collect*) por ejemplo, una vez recolectados los datos en el nodo master/driver (por medio de las operaciones count o collect) se transforman estos a un tipo de dato/objeto acorde al lenguaje que se esté trabajando, o se transforman a un tipo de dato estructurado (JSON, XML, YAML) y luego se escriben en un sistema de almacenamiento externo (por ejemplo, por medio de la acción copyToHadoop).

Las acciones activan el scheduler de Spark, que crea un grafo acíclico dirigido (llamado DAG), basado en las dependencias entre las transformaciones de los RDDs. En otras palabras, Spark evalúa una acción mirando las transformaciones establecidas para definir la serie de pasos que debe procesar para producir el resultado final de cada RDD. Luego, usando esta serie de pasos, llamado execution plan (o plan de ejecución), el scheduler genera las tareas para cada nodo ejecutor.

Por lo tanto, es mejor pensar que cada RDD consiste en instrucciones que determinan cómo calcular los datos que se construyen a través de transformaciones.

Spark utiliza la evaluación lazy para reducir el número de iteraciones (pasadas) sobre los datos agrupando operaciones de transformación. Es decir, que la evaluación diferida le permite a Spark combinar operaciones que no requieren comunicación con el driver (llamadas transformaciones con dependencias *one-to-one*) para evitar hacer múltiples pasadas a través de los datos. Por ejemplo, supongamos que un programa Spark llama a una transformación map y a una transformación filtro en el mismo RDD. Spark puede enviar simultáneamente, las instrucciones tanto de map como de filtro a cada ejecutor. Entonces Spark puede realizar a la vez tanto el map como el filtro en cada nodo ejecutor, lo que requiere acceder a los registros solo una vez, en lugar de enviar dos conjuntos de instrucciones y acceder a cada partición dos veces.

El paradigma de evaluación lazy de Spark no solo es más eficiente, sino que también es más fácil implementar la misma lógica en Spark que en un framework diferente, como Hadoop MapReduce, que requiere que el desarrollador haga el trabajo para consolidar sus operaciones de mapeo, es decir, a menudo tienen que pasar mucho tiempo considerando cómo agrupar operaciones para minimizar el número de iteraciones. La estrategia de evaluación de Spark nos permite expresar la misma lógica en muchas menos líneas de código: podemos encadenar operaciones con dependencias one-to-one como las de map y

filter, y dejar que el motor de evaluación Spark haga el trabajo de consolidarlas, permitiendo a los desarrolladores organizar su programa en operaciones más pequeñas y manejables.

Persistencia y administración de memoria

La mayor ventaja de rendimiento de Spark sobre MapReduce se da en los casos que implican realizar cálculos repetidos sobre los mismos datos. Gran parte de la ventaja en el rendimiento se debe a que Spark utiliza persistencia de datos sobre la memoria principal. En lugar de escribir en el disco entre cada operación, Spark tiene la opción de mantener cargados los datos en la memoria de los ejecutores. De esta forma, los datos en cada partición están disponibles en memoria cada vez que se necesita acceder a ellos, lo cual reduce considerablemente los tiempos de lectura.

Spark ofrece tres opciones para la gestión de memoria: 'en memoria como datos deserializados', 'en memoria como datos serializados' y 'en disco'. Cada uno tiene diferentes ventajas de espacio y tiempo:

- **En memoria como objetos Java deserializados:** La forma más intuitiva de almacenar objetos en RDD en el formato de los objetos originales deserializados de Java definidos por las clases en el programa driver. Esta forma de almacenamiento en memoria es la más rápida, ya que reduce el tiempo de serialización, por lo que los objetos siempre están listos para operar con ellos; sin embargo, puede que no sea la más eficiente desde el punto de vista de la memoria, ya que requiere que los datos se almacenen como objetos con todos sus atributos y con la representación de todos los tipos de estos atributos, lo cual trivialmente ocupa más espacio.

- **Como datos serializados:** Utilizando la biblioteca estándar de serialización de Java, los objetos Spark se convierten en flujos de bytes a medida que se mueven por la red. Este enfoque puede ser más lento, ya que los datos serializados consumen más CPU que los datos deserializados, debido a que tienen que pasar por la etapa de deserialización; sin embargo, a menudo es más eficiente desde el punto de vista de la memoria, ya que permite al usuario elegir una representación más eficiente y compacta. Si bien la serialización de Java es más eficiente que los objetos completos, la serialización de Kryo, la cual es soportada por Spark, puede ahorrar aún más espacio.

- **En disco:** Los RDD, cuyas particiones son demasiado grandes para almacenarse en RAM en cada uno de los ejecutores, se pueden escribir en el disco. Esta estrategia es obviamente más lenta para cálculos repetidos, pero puede ser más tolerante a

fallas para largas secuencias de transformaciones, y puede ser la única opción factible para cálculos enormes.

La función *persist()* de un RDD le permite al usuario controlar cómo se almacena el RDD. De forma predeterminada, *persist()* almacena un RDD como objetos deserializados en la memoria, pero el usuario puede pasar una de las opciones de almacenamiento a la función *persist()* para controlar cómo se almacena el RDD.

Tolerancia a Fallos

Spark es tolerante a fallas, lo que significa que Spark no fallará, perderá datos ni arrojará resultados inexactos en caso que se produzca una falla en una máquina del cluster. Esto se logra debido a que cada partición de un RDD contiene la información de las dependencia de transformaciones del RDD, las cuales se utilizan para calcular o recalcular la partición en caso de falla. La mayoría de los paradigmas computacionales distribuidos que permiten a los usuarios trabajar con objetos mutables proporcionan tolerancia a fallas al registrar actualizaciones o duplicar datos entre máquinas.

Por el contrario, Spark no necesita mantener un registro de las actualizaciones de cada RDD o registrar los pasos intermedios reales, debido a las ventajas de la evaluación lazy y a la inmutabilidad de un RDD, siempre se puede replicar la información de una partición. Además Spark permite que ese cálculo se puede paralelizar para acelerar la recuperación.

Anatomía de una aplicación en Spark

En el paradigma de evaluación lazy, una aplicación Spark no "hace nada" hasta que el programa del driver llama a una acción. Con cada acción, el planificador Spark crea un grafo de ejecución (DAG) y comienza un ***job*** (trabajo) de Spark. Cada **job** consta de ***stages*** (etapas), que son las operaciones de transformación que se especifican en cada RDD para materializar el RDD final. Cada **stage** consiste en una colección de **task** (tareas) que representan cada cómputo paralelo y son realizadas en los nodos ejecutores.

Para tener una idea más concreta de cómo interactúan los diferentes componentes de una aplicación Spark, a continuación la figura 2.3 representa el árbol de los diferentes componentes y cómo estos corresponden con las llamadas a la API de Spark.

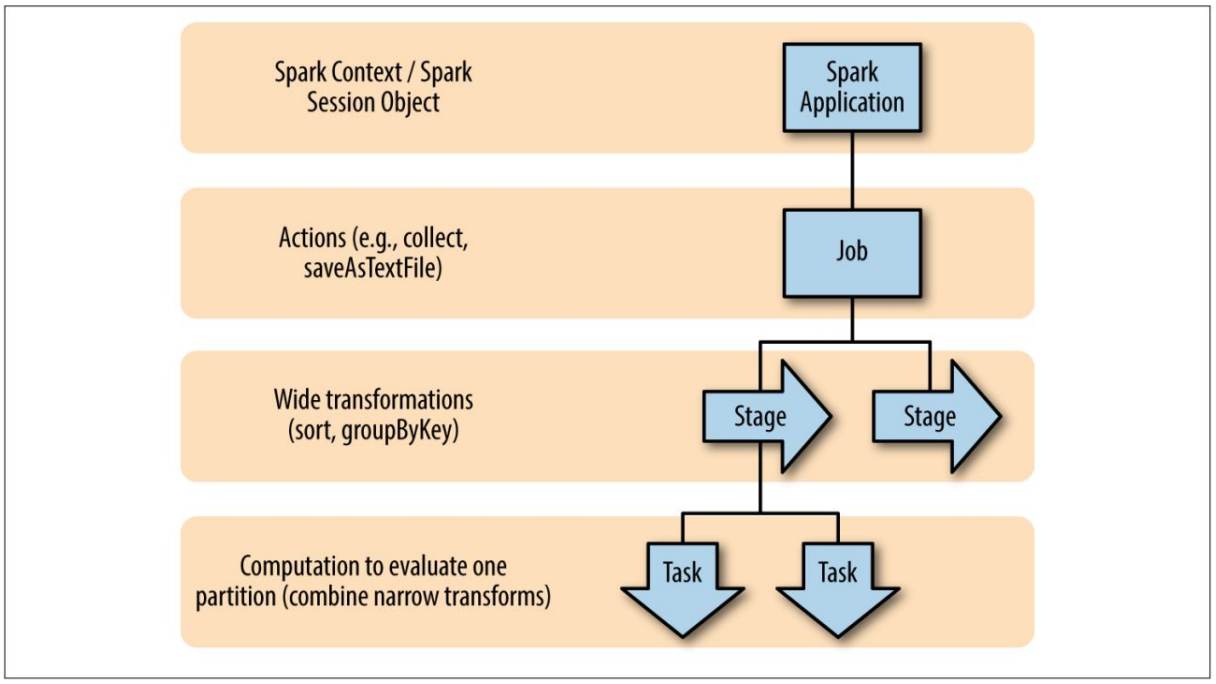


Figura 2.3: Árbol de componentes de Spark. Figura extraída de [30]

Como se ve en el gráfico, una aplicación spark inicia cuando se crea o asigna un SparkContext/SparkSession. Cada aplicación puede contener muchos jobs (trabajos) que corresponden a cada una de las acciones definidas sobre un RDD, es decir, que la relación entre una acción y un Job es uno a uno. Cada job puede contener varios stages (etapas) que corresponden a cada transformación, al igual que en los jobs-acciones, la relación entre una transformación y un stage es uno a uno, pero entre job-stage es uno a muchos. Cada stage se compone de una o muchas task (tareas) que corresponden a una unidad de cálculo paralelizable computada por los nodos ejecutores, entonces la relación stage-task, es uno a muchos.

DAG

Hasta el momento, se ha mencionado que es el DAG pero no se lo ha definido concretamente ni se ha explicado cómo interactúa con los componentes mencionados anteriormente, es por esto, que esta sección se encarga de dar una explicación de este componente.

Spark utiliza un capa de scheduling de alto nivel encargado de construir un grafo acíclico dirigido (DAG) de stage para cada Job de Spark, es decir, que es el encargado de transformar las operaciones de transformación en stage luego de que se invoca una acción en el driver. En la API de Spark, esto se llama como DAG Scheduler. Por lo tanto, El DAG se encarga de manejar/administrar la ejecución de los jobs Spark. El DAG, luego de crear un

grafo de stage para un Job, determina las ubicaciones para ejecutar cada tarea en el cluster y pasa esa información al TaskScheduler, que es responsable de enviar y ejecutar las tareas sobre la particiones de un RDD que posee un nodo ejecutor.

En el siguiente ejemplo se muestra cómo se genera un Job (figura 2.5) cuando un programa simple escrito utilizando la API de Spark (figura 2.4) es ejecutado:

```
def simpleSparkProgram(rdd : RDD[Double]): Long ={
//stage1
  rdd.filter(_< 1000.0)
     .map(x => (x, x) )
//stage2
     .groupByKey()
     .map{ case(value, groups) => (groups.sum, value)}
//stage 3
     .sortByKey()
     .count() //Action
}
```

Figura 2.4: Ejemplo de programa utilizando Spark

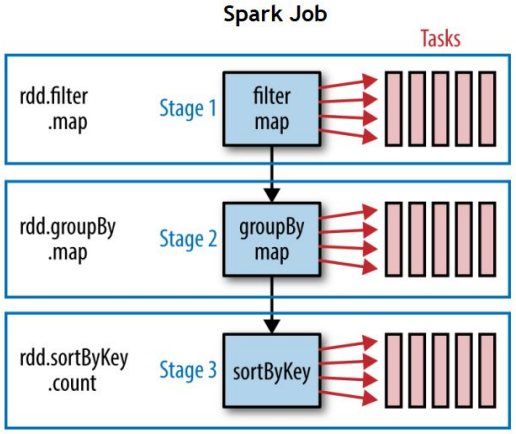


Figura 2.5: Resultado de generar un Job a partir del programa anterior.

Spark Streaming

Muchas aplicaciones necesitan procesar los datos tan pronto como llegan. Por ejemplo, una aplicación puede rastrear estadísticas sobre visitas de página en tiempo real, entrenar un modelo de aprendizaje automático o detectar anomalías automáticamente. Spark Streaming es el módulo de Spark para tales aplicaciones. Permite a los usuarios escribir aplicaciones sobre un flujo de datos utilizando una API muy similar a los batch jobs definidos en el core del framework, y así reutilizar muchas de las funcionalidades de esa API como por ejemplo las transformaciones.

Al igual que Spark, Spark Streaming se basa en el concepto de RDD, Spark Streaming ofrece una abstracción llamada **DStreams** o **flujos discretizados**. Un DStream es una secuencia de datos, donde estos llegan a medida que pasa tiempo. Internamente, cada DStream se representa como una secuencia de RDDs que llegan en un intervalo de tiempo (de ahí el nombre "discretizado"). Ofrecen dos tipos de operaciones: transformaciones, que producen un nuevo DStream, y operaciones de salida, que escriben datos en un sistema externo. DStreams proporciona muchas de las mismas operaciones disponibles en los RDD, más nuevas operaciones relacionadas con el tiempo, como ventanas deslizantes.

Arquitectura y Abstracción sobre Spark

Spark Streaming utiliza una arquitectura de "micro-batch", donde el procesamiento del flujo de datos se trata como una serie continua de computaciones sobre pequeños batch (lotes) de datos. Spark Streaming recibe datos de varias fuentes de entrada y los agrupa en pequeños batch. Los batch se crean a intervalos de tiempo regulares. Al comienzo de cada intervalo de tiempo, se crea un nuevo batch y todos los datos que llegan durante ese intervalo se agregan a ese batch. La duración de los intervalos de tiempo es un parámetro configurable de la aplicación Spark. Generalmente, el intervalo suele ser de entre 500 milisegundos y varios segundos, según lo configurado por el desarrollador de la aplicación. Cada batch de entrada forma un RDD y se procesa mediante los Jobs Spark para crear otros RDD. Los resultados procesados luego pueden enviarse a sistemas externos. Esta arquitectura de alto nivel se muestra en la figura 2.6.

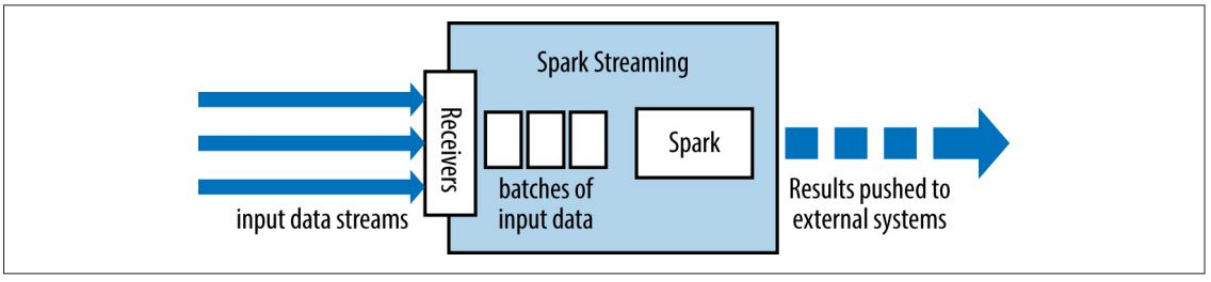


Figura 2.6: Arquitectura Spark Streaming. Figura extraída de [30]

Como ya se mencionó, la abstracción de programación en Spark Streaming es una secuencia discretizada o DStream, que es una secuencia de RDDs, donde cada RDD representan los datos en una porción o intervalo de tiempo.

Figura 2.7: Representación de un flujo de datos DStream. Figura extraída de [30]

Al igual que un RDD, se puede crear un DStream desde fuentes de entrada externas o aplicando transformaciones a otras DStream, es decir, que también son inmutables. DStream admite muchas de las transformaciones que admiten los RDD, pues las transformaciones dentro de la API de DStream son simplemente un wrapper de las transformaciones sobre RDDs.

Además de las transformaciones, DStream admite operaciones output (de salida). Las operaciones de salida son similares a las acciones RDD, por ejemplo escriben datos en un sistema externo. Cada operación de output en Spark Streaming se ejecuta periódicamente en cada finalización del intervalo de tiempo definido para los micro-batch, y por ende, produciendo resultados en batchs.

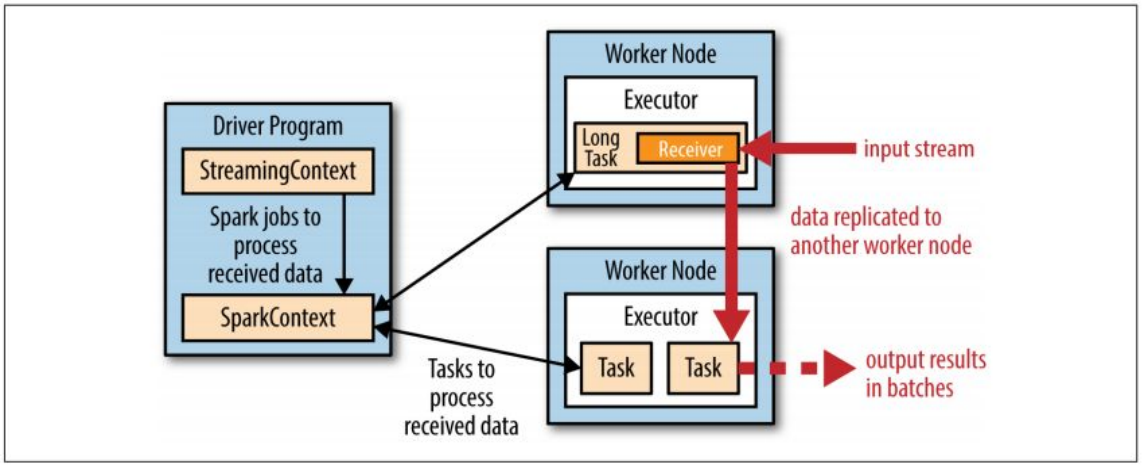


Figura 2.8: Ejecución de Spark Streaming mediante los componentes del core de Spark. Figura extraída de [30]

La ejecución de Spark Streaming (Figura 2.8) dentro de los componentes driver-worker de Spark se realiza de la siguiente forma: Para cada *input source* (fuente de entrada), Spark Streaming lanza *receivers* (receptores), que son tasks que se ejecutan dentro de los nodos ejecutores de la aplicación y se encargan de recopilar datos de la fuente de entrada y guardarlos como RDD, además se replican los datos entre los nodos ejecutores con el objetivo de mantener la propiedad de tolerancia a fallas. Luego, el StreamingContext del programa driver ejecuta periódicamente las tasks de los Jobs Spark definidos por el usuario

para procesar estos datos y si es necesario, combinarlos con los RDD de intervalos anteriores.

API DStream

Para cerrar este capítulo, a continuación se mencionan las característica de la API de Spark para el manejo de flujo de datos.

La API DStream de Spark Streaming se basa en la API de RDD, por lo que la mayoría de los operaciones son wrappers a los métodos definidos en la API de RDD y por lo tanto, tienen las mismas características de rendimiento. Pero más allá de las operaciones 'heredadas' de los RDD, los DStreams definen nuevas operaciones las cuales se describen a continuación.

Transformaciones

Las transformaciones para la API de DStream se pueden agrupar en 2 categorías:

Transformaciones sin estado (*stateless transformations*): el procesamiento de cada batch no depende de los datos en los batchs anteriores. Por lo tanto, esta categoría incluye las transformaciones RDD comunes que se han visto para los RDD. Además se agrega una transformación nueva, la cual recibe el nombre de **`transform`** y permite operar directamente sobre los RDD dentro de cada DStream. La operación `transform()` recibe como parámetro una función del tipo RDD => RDD, y devuelve un nuevo DStream que es el resultado de aplicar la función parámetro sobre cada RDD del flujo original. Una aplicación común de `transform` es reutilizar código que se haya implementado para el procesamiento de los RDD en la API tradicional.

Transformaciones con estado (*stateful transformations*): estas transformaciones usan datos o resultados intermedios de batchs anteriores para calcular los resultados del batch actual. Dentro de estas transformaciones se encuentra la transformación `window` que permite realizar cálculos sobre los últimos K batch del flujo, es decir, lleva a cabo un procesamiento en un período de tiempo más largo que el intervalo de batch (*batchInterval*) determinado por el StreamingContext al combinar los resultados de varios batchs.

Las operaciones `windows` necesitan dos parámetros, *windowDuration* y *slidingDuration*, que deben ser múltiplos del intervalo del batch definido en el StreamingContext. *windowDuration* controla cuántos batch de datos previos se procesaran, es decir, los últimos (*windowDuration* / *batchInterval*) batch. Si se tuviera un DStream con un intervalo de batch de 10 segundos y se quiere crear una ventana deslizante de los últimos 30 segundos (o los

últimos 3 lotes), entonces se tiene que establecer la *windowDuration* en 30 segundos. El argumento *slidingDuration* que por defecto es el intervalo del batch, controla la frecuencia con la que el nuevo DStream calcula los resultados. Por ejemplo, si se tiene un DStream con un intervalo de 10 segundos y se quiere calcular una ventana luego de recibir los últimos dos batch, se tiene que establecer *slidingDuration* en 20 segundos.

Operaciones Output

Como ya se mencionó anteriormente las operaciones de Output son operaciones similares a las acciones RDD y por ende, especifican lo que se debe hacer con los datos del flujo luego de computar todas las transformaciones anteriores (por ejemplo, enviarlos a una base de datos externa o imprimirlo en la pantalla).

También, al igual que la evaluación lazy en los RDD, si no se aplica una operación de output en un DStream y cualquiera de sus descendientes, estos DStreams no se evaluarán. Y si no hay operaciones de output establecidas en un StreamingContext, entonces el contexto no se iniciará.

A diferencia de las acciones en los RDD, la API de DStream para las operaciones de output es más limitada, por ejemplo, se pueden realizar solo dos operaciones output específicas, imprimir los primeros diez elementos en el nodo master y guardar los elementos del flujo en archivos de texto plano.

Pero para solucionar esta limitación Spark Streaming brinda una operación de output genérica llamada `foreachRDD()`. Esta operación nos permite ejecutar cualquier cálculo directamente sobre los RDD del DStream. `foreachRDD()` es similar a `transform()` ya que da acceso y permite trabajar con cada RDD. Dentro de `foreachRDD()`, se puede reutilizar todas las acciones que tenemos en Spark. Por ejemplo, un caso de uso común es escribir datos en una base de datos externa como MySQL. Además `foreachRDD()` es un método sobrecargado ya que también existen definiciones de `foreachRDD` que reciben como parámetro información de contexto como por ejemplo el timestamp del RDD/batch actual.

Capítulo 3

Clustering - Agrupamiento

El clustering, o técnica de agrupación, es una tarea clave dentro del área la Minería de datos (Data Mining). Es el proceso de particionar, fraccionar, repartir o dividir un conjunto de observaciones en distintos grupos o subconjuntos (clusters), de manera que las observaciones dentro de un grupo (intra-cluster) sean similares y las observaciones entre grupos (inter-cluster) sean diferentes, con respecto a una medida de similitud dada.

Los objetivos principales de las técnicas de clusterings son:

- Clasificar objetos en grupos de objetos similares.
- Reducir la cantidad de datos a partir de reemplazar todos los datos similares dentro de cada grupo por un nuevo valor o modelo que se encargue de representar (o abstraer) las características generales del grupo.
- Buscar patrones en el conjunto de datos.

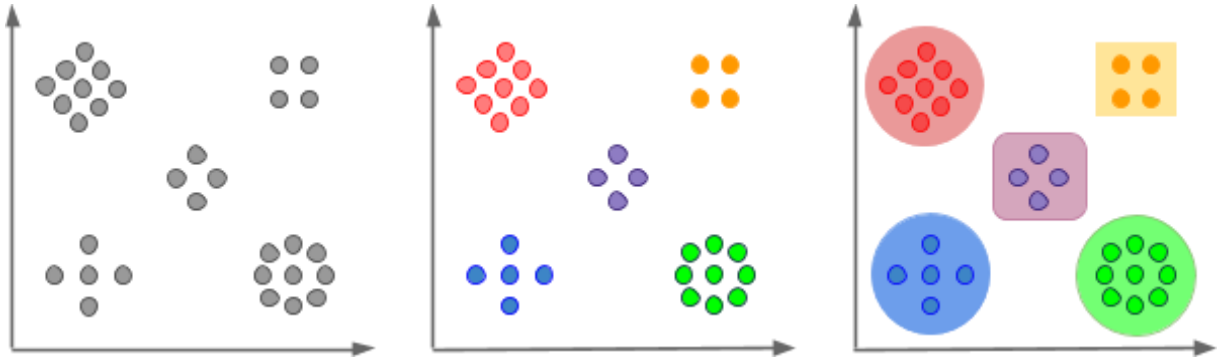


Figura 3.1: Ejemplo de Agrupamiento.

Data Stream Clustering

El set-up o la organización tradicional en la que el conjunto de datos estáticos está disponible en su totalidad para el acceso aleatorio y sobre la cual se puede realizar lecturas repetidamente, no es aplicable a los flujos de datos, ya que no tenemos todo el conjunto de datos en el momento que ejecutamos esta técnica, esto se debe a las características que se nos presentan en el contexto de los flujos de datos, las cuales se vienen mencionando en este trabajo: los datos llegan a un ritmo rápido, no podemos acceder a los datos aleatoriamente, y solo podemos hacer una o como mucho un número pequeño de pasadas sobre los datos para generar los resultados de la agrupación.

El problema de agrupamiento de flujo de datos, o "data stream clustering" (como se menciona a este problema en la literatura), requiere un proceso capaz de detectar distintos grupos de forma continua teniendo en cuenta las restricciones de memoria y tiempo. En la literatura sobre los métodos de agrupamiento de flujo de datos, una gran cantidad de algoritmos usan un esquema o enfoque de dos fases que consiste en tener un componente on-line que procesa los elementos actuales o recientes del flujo de datos y producir un resumen estadístico. Y por otro lado, un componente offline que usa los datos de resumen para generar los grupos (clusters). Un Enfoque alternativo, es utilizar solamente el componente online, el cual tiene que ser capaz de generar los clusters finales sin la necesidad de una fase o componente offline.

Al aplicar técnicas de minería de datos, y específicamente algoritmos de clustering, sobre flujos de datos, las restricciones de tiempo de ejecución y de la memoria deben considerarse cuidadosamente. Para lidiar con estas restricciones, muchos de los algoritmos de cluster de flujo de datos existentes modifican los métodos tradicionales de clustering (non-streaming) para usar el enfoque de dos fases propuesto en [5] para tratar los distintos elementos que derivan de los flujos de datos, por ejemplo, DenStream[14] es una extensión del Algoritmo DBSCAN [35], StreamKM++ [36] es una extensión de k-means++[37], StrAP [38] es una extensión de AP [38], etc.

A continuación se presenta una reseña de los métodos de agrupación de flujos de datos más importantes dentro del estado del arte de tema. Se describirán para cada técnica, los conceptos y características más importantes que utilizan como así también se realizará un análisis de las desventajas que pueden presentar.

BIRCH

Balanced Iterative Reducing and Clustering using Hierarchies
(Reducción iterativa equilibrada y agrupamiento utilizando jerarquías)

Es un algoritmo de clustering que genera de forma incremental clusters en un solo scan de los datos multidimensionales intentando producir el mejor resultado dependiendo de los recursos disponibles. Es por esto, que BIRCH, puede trabajar con datasets muy grandes ya que primero genera resúmenes de los datos, llamados Clustering Feature (Característica de Agrupamiento) y luego utiliza estos resúmenes para generar grupos o clusters. Siguiendo esta metodología, podemos deducir que primero se hace un único análisis o procesamiento sobre los datos originales y luego sigue su ejecución con el resultado de los resúmenes generados. Luego, se puede mejorar la calidad de los clusters generados, realizando nuevas pasadas sobre los datos, siempre que estos estén disponibles y haya también recursos disponibles.

A pesar de que BIRCH, no fue pensado para trabajar sobre flujos de datos, esta propiedad es especialmente útil en el caso de los flujos de datos, ya que esencialmente los datos entrantes del flujo generalmente no se almacenan, debido a que pueden ser potencialmente infinitos, y por lo tanto, múltiples pasadas sobre los datos generalmente requieren mucho tiempo o simplemente no son posibles.

Posee dos estructuras especiales para la ejecución del método de agrupación, las cuales los algoritmos posteriores a éste usan como base para sus respectivos enfoques:

Clustering Feature (CF):

La definición de características de cluster o cluster-feature, extraída del paper de BIRCH[45] es la siguiente:

> CF Definition: Given N d-dimensional data points in a cluster: $\{\overline{X_i}\}$ where $i = 1, 2, ..., N$, the **Clustering Feature (CF)** entry of the cluster is defined as a triple: $CF = (N, \overline{LS}, SS)$**,** where N is the number of data points in the cluster, $\overline{LS}$ is the linear sum of the N data points, i.e $\sum_{i=1}^{N} \overline{X_i}$, and SS is the square sum of the N data points, i.e, $\sum_{i=1}^{N} \overline{X_i}^2$.

Es decir, dado un conjunto de N datos de d-dimensiones un Clustering Feature se define como una 3-tupla CF = (N, LS, SS) donde:

- N: es el número total de datos en el conjunto.
- LS: es la suma lineal de los N elementos del conjunto de datos. En este punto, podemos ver que en la definición de LS hay una característica que puede pasar por desapercibida, por lo tanto se hace la siguiente observación: LS, es un vector de tamaño d, donde d es el número de dimensiones que poseen los datos de entrada, por lo tanto, LS es un vector, que posee las sumas lineales de cada dimensión de los datos de entrada en el conjunto o grupo.
- SS: es la suma cuadrada de los N elementos del conjunto de datos. Al igual que LS, SS es un vector de tamaño d, que posee las sumas cuadrada de cada dimensión de los datos de entrada en conjunto o grupos de datos.

A parte de esto, los CF tiene las siguiente propiedad/teorema:
Teorema 1 (additivity): Sea $CF_1 = (N_1, \overline{LS}_1, \overline{SS}_1)$, sea $CF_2 = (N_2, \overline{LS}_2, \overline{SS}_2)$ y además CF_1 y CF_2 son dos clustering feature de subconjuntos disjuntos, entonces el CF resultante de fusionar los dos subconjuntos disjunto es igual a:

$$CF_R = CF_1 + CF_2 = (N_1 + N_2, \overline{LS}_1 + \overline{LS}_2, \overline{SS}_1 + \overline{SS}_2)$$

Por ejemplo, supongamos que hay tres puntos (2, 3), (4, 5), (5, 6) en el subgrupo C1, entonces el CF de C1 es:

$$CF_1 = \{3, (2+4+5, 3+5+6), (2^2+4^2+5^2, 3^2+5^2+6^2)\}$$
$$CF_1 = \{3, (11, 14), (45, 70)\}$$

Supongamos que hay otro clúster C2 con CF2 = {4, (40, 42), (100, 101)}. Luego, el CF del nuevo clúster formado por la fusión del clúster C1 y C2 es:

$$CF_3 = \{3 + 4, (11 + 40, 14 + 42), (45 + 100, 70 + 101)\}$$
$$CF_3 = \{7, (51, 56), (115, 171)\}$$

La demostración del teorema consiste en el álgebra del espacio vectorial convencional[39]. De acuerdo con el teorema de adición de CF, las entradas de CF se pueden almacenar y calcular incremental y consistentemente a medida que se fusionan subgrupos o se insertan nuevos puntos de datos.

A parte de esto, los CF no sólo son compactos pues almacenan mucha menos información que lo que ocupa guardar todos los datos del subgrupo, también son precisos ya que es suficiente la información que tienen para calcular todas las propiedades y medidas que

necesitamos para hacer clustering en BIRCH, como por ejemplo, calcular el centroide y radio de un CF:

$$\text{Centroide: } C = \frac{\sum_{i=1}^{N} x_i}{N} \quad \text{- Radio: } R = \sqrt{\frac{\sum_{i=1}^{N} \| x_i - C \|^2}{N}}$$

CF Tree

Un CF tree es un árbol balanceado con 2 parámetros:

1. Un Factor de ramificación B para los nodos internos (no hojas) y un factor L para los nodos hojas.
2. Un umbral o límite T.

Cada nodo interno contiene como máximo B entradas de la forma $[CF_i, hijo_i]$, donde $i = 1, 2, ..., B$. El $hijo_i$ es un puntero al i-ésimo nodo hijo en el árbol y CF_i es el clustering feature del i-ésimo hijo del nodo actual, el cual está formado por los subgrupos que se encuentran en el $hijo_i$.

Por lo tanto, un nodo interno o un nodo hoja, representa un sub-cluster compuesto por todos los subgrupos que se encuentran en sus hijos, y a su vez dicho sub-cluster se encuentra almacenado en su nodo padre.

Un nodo hoja puede tener como máximo L entradas donde todas las entradas son CF en sí mismas. Pero, al mismo tiempo, todas las entradas en un nodo hoja deben ajustarse a un umbral T, es decir, el diámetro o radio de cada entrada de hoja debe ser menor que T, lo cual establece el tamaño mínimo para un subcluster.

El árbol CF se construirá dinámicamente a medida que se inserten nuevos objetos de datos. A medida que llega un dato nuevo, se utiliza el árbol para guiar la nueva inserción en el subcluster correcto (en base al subgrupo más cercano), de la misma manera que se usa un árbol B+. El árbol CF es una representación muy compacta del conjunto de datos porque cada entrada en un nodo hoja no es un punto de datos único sino un subgrupo (que absorbe tantos puntos de datos como lo permita el valor umbral especificado).

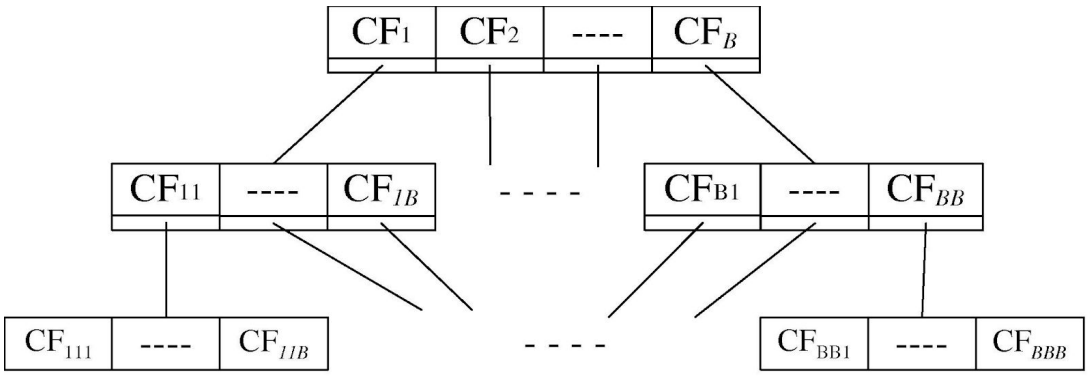


Figura 3.2: Ejemplo de Árbol CF con altura 3

Tras definir la estructura del árbol CF, se establecen los pasos a seguir para el proceso de inserción de datos o un subgrupos de datos en dicho árbol:

1. **Identificación de hojas:** Para los datos de entrada, primero se busca el nodo hijo más cercano.

2. **Modificación de hojas:** Los datos o los subcluster de entrada son absorbidos por la hoja más cercana (nearest leaf) siempre y cuando no viole el umbral T. Si el umbral T es sobrepasado, se crea una nueva hoja en el nodo padre si es que este último contiene el espacio suficiente para almacenar un nuevo nodo. Si el nodo padre no cuenta con el espacio suficiente, la hoja nearest leaft, es dividida en 2 nuevas hojas, que son lo mas lejanas entre sí. Luego estas se fusionan en las hojas que cumplan un determinado criterio, por ejemplo, se fusionan en las hojas más cercanas.

3. **Modificación del camino de una hoja:** Cuando se inserta una nueva entrada a un nodo hoja, se actualizan cada uno de los CF de los nodos internos que existen en el camino desde la raíz a la hoja. Este proceso es simple si la nueva entrada simplemente se absorbe en una hoja, pero en el caso en que se genere una nueva hoja o el caso en que una hoja se divide en dos, la información sobre el nuevo nodo hoja debe insertarse en el nodo padre. Esta nueva entrada en el nodo principal no hoja podría a su vez causar una división si no hay suficiente espacio disponible en el nodo interno. Este comportamiento en general podría propagarse hasta la raíz, y si la raíz necesita dividirse también, en ese caso la altura del árbol aumenta en 1.

4. **Fusión de refinamiento o reajuste**: las divisiones son causadas por el tamaño que se asignan a los nodos, que es independiente de las propiedades de agrupación de los datos. En presencia de un orden de entrada de datos asimétrico (o datos sesgados), esto puede afectar la calidad del clúster y también reducir la utilización del espacio. Para resolver este problema, se puede realizar una combinación o

fusión adicional a partir del nodo interno donde finaliza la propagación de división de hojas. Para esto se escanea dicho nodo para encontrar las dos entradas más cercanas. Si no son el par correspondiente a la división, tratamos de fusionarlos y luego se tratan de fusionar los nodos secundarios correspondiente. Por lo tanto, la combinación entre 2 nodos también da como resultado una posible fusión de los nodos secundarios correspondientes, a parte de esto, liberamos un nodo para su uso posterior y creamos espacio para una entrada más, aumentando así la utilización del espacio y posponiendo las divisiones futuras; de este modo, mejoramos la distribución de las entradas en los dos hijos más cercanos.

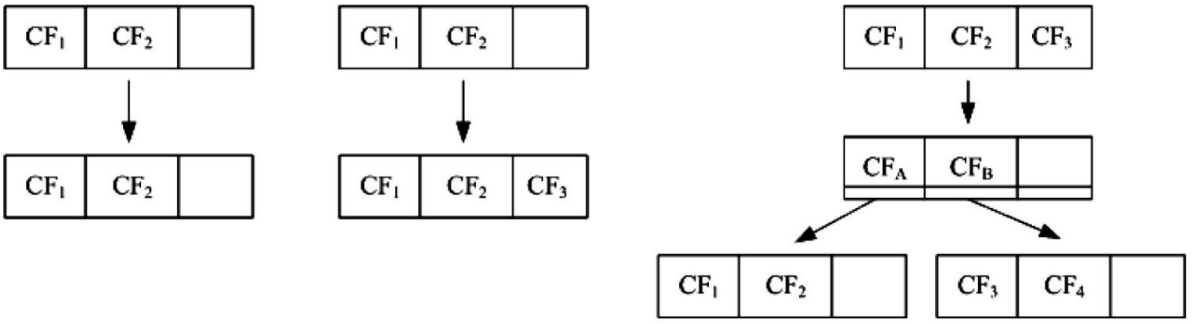


Figura 3.3: Formas de inserción en un Árbol CF.

Algoritmo BIRCH

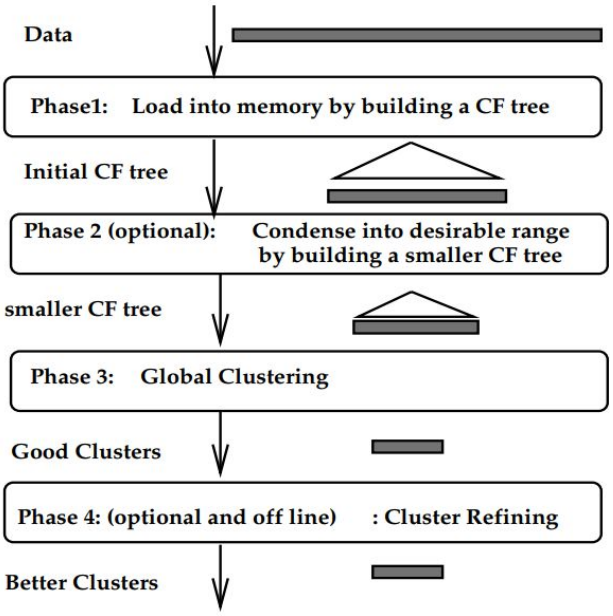


Figura 3.4: algoritmo BIRCH.

El algoritmo de agrupación de BIRCH consiste en 4 etapas, algunas de las cuales son opcionales y se utilizan únicamente para mejorar los resultados. Sin embargo, es importante tener en cuenta que estos pasos opcionales puede requerir pasadas o iteraciones adicionales sobre los datos.

1. **Carga (Loading):** esta fase escanea todos los datos entrantes y crea en memoria el árbol CF inicial, sujeto a la memoria disponible. Es decir, esta fase se encarga de crear esencialmente el resumen de los datos entrantes. Esta es la fase más importante de BIRCH, ya que reduce el problema de la agrupación sobre todos los datos entrantes a agrupar subgrupos o subcluster que se encuentran presentes en las hojas del árbol. Esto es importante porque agrupar los subgrupos es esencialmente mucho más rápido que realizar agrupamientos sobre todos los datos presentes en el flujo. Además, esta fase también elimina los valores atípicos, por lo que ayuda a generar clusters resultantes más precisos.

2. **Condensación (Condensing)**: esta fase es opcional y realiza la condensación o reducción del árbol de CF inicial mediante la reconstrucción de un árbol de CF más pequeño. Esto se consigue eliminando más valores atípicos y juntando (condensando) los subgrupos que están muy juntos en otros más grandes.

3. **Agrupación global (Global Clustering):** esta fase realiza la agrupación del árbol CF resultante de la fase 1 o después de la condensación de la fase 2. De este modo, se obtienen los diferentes patrones y estructuras disponibles que se encuentran escondidos en los datos. Para realizar el proceso de agrupación BIRCH utiliza una adaptación de un algoritmo de agrupamiento jerárquico aglomerativo paralelo descrito en [40]. También, existen implementaciones de BIRCH que adaptan los algoritmos de CLARANS y KMEANS para que puedan agrupar CF en vez de puntos, y de esta forma utilizarlos para llevar a cabo la agrupación global mediante los nodos del árbol.

4. **Refinación (Refining):** esta fase también es opcional pero tiene como objetivo “afinar” los clusters creados en la fase 3 utilizando los centros de cluster como semillas y luego redistribuyendo los puntos de datos a los centros de cluster más cercanos, obteniendo así un resultado refinado y un mejor conjunto de clusters. Se debe tener en cuenta, que esta etapa realiza otra pasada sobre los datos, lo que generalmente no es posible con flujos puros e infinitos.

Problemas con CF Tree:

La limitación de un número fijo de entradas por nodo introduce 2 desafíos principales en BIRCH:
Dos subclusters que deberían estar en el mismo cluster se dividen en nodos diferentes y, de forma similar, dos subclusters que no deberían estar juntos son parte del mismo nodo, esto también se produce debido al orden en el que llegan los datos, no solo al tamaño del nodo.

En caso de que el mismo dato se inserte dos veces, el cluster resultante para ambos podría ser diferente, ya que podrían colocarse en diferentes nodos. Este desafío se resuelve mediante el uso de la fase 4 del algoritmo BIRCH donde los clústeres se refinan reajustando los datos según los clusters. Por lo tanto, este desafío no se puede resolver fácilmente porque los datos no están disponibles para una segunda pasada en escenarios de flujos de datos infinitos y con tasas de velocidad alta.

Características importantes y limitaciones

Características Importantes

- Es el primer algoritmo que propuso almacenar información resumida como características de un grupo/cluster (CF). Donde esta metodología es utilizada ampliamente en otros algoritmos.
- Puede manejar una gran cantidad de datos.
- Luego de generar el árbol de CFs, es posible aplicar cualquier algoritmo de clustering sobre los datos que contiene.

Limitaciones

- Para mejorar la calidad del agrupamiento, se necesita obligatoriamente realizar una segunda lectura de los datos, lo cual, en muchos casos es imposible por la naturaleza del flujo.
- No detecta Concept Drift.
- No contempla el envejecimiento "aging" de los datos.

ClusTree

Según los autores, es un algoritmo de clustering que se enfrenta al desafío de mantener siempre un resultado actual que pueda presentarse al usuario en cualquier momento dado. En este trabajo, se presenta el primer algoritmo bajo el enfoque de “agrupamiento en cualquier momento” (o en ingles, “Anytime clustering approach”), donde se propone un algoritmo sin parámetros (parameter-free) que se adapta automáticamente a la velocidad del flujo de datos, limitándose a la memoria disponible y detectando concept-drift y valores atípicos. A parte de esto, se incorpora el concepto de “la edad de los objetos” (o en ingles “the age of the objects”) para reflejar la importancia de los datos más recientes.

Para un manejo eficiente y efectivo, ClusTree, usa una estructura de índice compacta y autoadaptativa para mantener los resúmenes de subgrupos del flujo. Con respecto a la autodaptatividad, ClusTree, se adapta dinámicamente tanto al número de clusters como a la velocidad del flujo de datos mediante el uso de un concepto llamado *inserciones en cualquier momento (anytime inserts), lo cual permite que* en cualquier momento, ClusTree sea capaz de entregar un resultado, y cuando haya tiempo disponible utilizarlo para refinar el resultado. Esto significa que el algoritmo es capaz de procesar incluso flujos muy rápidos, pero luego se necesitará y utilizará una gran cantidad de tiempo disponible para refinar el modelo de agrupamiento.

En cuanto al número de clusters (específicamente, el número de micro-clusters), Clustree no hace suposiciones a priori sobre el número de grupos en el modelo, por lo que no necesita un parámetro para indicar la cantidad de grupos a formar.

ClusTree: Micro Clusters e inserciones Anytime

ClusTree crea y mantiene resúmenes compactos de la secuencia de datos entrante utilizando la técnica popular conocida como Micro-Clusters que ya se explicó cuando se detalló el algoritmos BIRCH. Por lo tanto, en lugar de almacenar todos los objetos entrantes, mantiene una tupla de características de cluster (CF) similar a BIRCH, dada por CF = (n, LS, SS), la cual es suficiente para calcular centro, radio, varianza y además puede actualizar incrementalmente.

El problema con el enfoque de micro-clusters tradicional es que carece de soporte para administrar los CF cuando el flujo de datos cambia de velocidad. Por lo que los autores de

ClusTree proponen extender las estructuras de indexación disponibles de la familia de árboles R [41], [42], para mantener los CF's y tratar de brindar una solución a este problema. Esto permite mantener una jerarquía de microclusters a diferentes niveles de granularidad, siendo las hojas del árbol los elementos de granularidad más finas y los nodos intermedios tendrán una granularidad más gruesa cuanto más cerca se encuentren del nodo raíz. Dichas estructuras de indexación jerárquica proporcionan un mecanismo eficiente para localizar e insertar los nuevos datos de la secuencia en el micro-cluster correcto. Por lo tanto, si un dato determinado llega a un nodo de hoja y el dato es similar al micro-cluster, entonces es absorbido por él, de lo contrario se crea un nuevo micro-cluster.

El problema con esta jerarquía es que puede darse el caso en el que no haya tiempo suficiente para insertar el dato actual en el nodo hoja ya que la velocidad de la secuencia varía y el tiempo necesario para encontrar el micro-cluster adecuado puede ser mayor que la velocidad a la que llega el siguiente elemento.

Para solucionar esto, se propone el concepto de inserciones en cualquier momento (**Anytime Insertions)**: Para mantener la información necesaria para el proceso de agrupación y para garantizar a la vez, que cualquier objeto recién llegado se pueda insertar, se propone interrumpir el proceso de inserción. El elemento actual debe almacenarse temporalmente en un buffer local (local aggregate) que se encuentra en el último nodo intermedio del árbol indicado por el proceso de búsqueda antes de su interrupción, desde el cual se podrá continuar su inserción más adelante. Por lo que tendremos más demanda de espacio para todos los nodos intermedios pero con el espacio invertido, obtenemos una mayor precisión. Luego, cuando algunos de los siguientes procesos de inserción pueda llegar a este nodo intermedio, recuperará todas las entradas de los buffers y seguirá el proceso de inserción de estos junto con el que se encuentra llevando a cabo. Cabe recalcar, que si el proceso de inserción llegará a ser interrumpido, se utilizará la misma metodología de interrupción para todos los datos pendiente de inserción recolectados.

Definición de ClusTree:

Un ClusTree[43] con parámetros m, M de capacidad para los nodos intermedios y con parámetros l, L de capacidad para los nodos hoja, es una estructura de indexación multidimensional balanceada con las siguientes propiedades:

- Un nodo interno contiene entre m y M entradas. Los nodos hoja contienen entre l y L entradas. La raíz tiene al menos una entrada.
- Una entrada en un nodo interno de un ClusTree almacena lo siguiente:
    - Un micro-cluster CF de los objetos que resume (resumen a partir de los hijos).
    - Un micro-cluster CF de los objetos en el buffer (el cual puede ser vacío).

    - Un puntero al nodo hijo.
- Una entrada en una hoja de un ClusTree almacena un micro-cluster (CF) del objeto o de los objetos que representa. No contiene buffers.
- Una ruta desde la raíz a cualquier nodo hoja siempre tiene la misma longitud, ya que es un árbol balanceado.

El árbol definido anteriormente se crea y mantiene como cualquier árbol indexado multidimensional R, árbol R *, etc. Para la inserción, se elige el subárbol más cercano con respecto a la distancia euclidiana.

Aquí, es importante mencionar que el buffer en cada entrada del árbol es realmente crucial para mostrar la capacidad de las inserciones Anytime y además mostrar el potencial que tiene esta estructura. El buffer se utiliza como un almacenamiento temporal para los datos cuando el procedimiento de inserción se interrumpe y el dato no puede llegar al nodo hoja a tiempo, el CF actual simplemente se almacena en el búfer de la entrada que corresponde al subárbol en el que descenderá a continuación. Por lo tanto, cada vez que se realiza un acceso futuro a este subárbol, la entrada del búfer temporal se recupera y se continúa el proceso de inserción. Luego, cada vez que se baja por el subárbol, y el destino difiere para el dato original y el recuperado del buffer, este último se coloca en el búfer del nodo correspondiente (nodo diferente al actual), de modo que alguna otra inserción pueda recuperarlo para seguir su proceso.

Método para mantener actualizado los clusters

Para mantener una visión actualizada de los grupos formados hasta el momento, se necesita que los objetos nuevos sean más importantes que los objetos más antiguos. Una solución común es agregar un peso (ponderar) a los objetos por medio de una función de envejecimiento exponencial dependiente del tiempo

$$w(t) = \beta^{-\lambda \Delta t}$$

La tasa de descomposición λ controla cuánto peso tienen los artículos nuevos sobre los viejos, en otras palabras, controla cuánto más se favorece a los objetos nuevos en comparación con los antiguos. Cuanto mayor es λ, más rápido el algoritmo "olvida" los datos antiguos. Los autores de ClusTree establecen que β tiene que establecerse β = 2 como valor óptimo, de esta forma, la vida media de los objetos es $\frac{1}{\lambda}$.

Para incorporar la descomposición o envejecimiento, se debe agregar información relacionada con el tiempo en los nodos del árbol de ClusTree. Lo cual garantiza que los

nodos internos del árbol resumen sus subárboles correspondientes de forma más precisa al lograr que los elementos de un micro-cluster (CF) dependan de (o den más importancia) los datos más reciente en el tiempo actual t, por lo tanto:

$$n^{(t)} = \sum_{i=1}^{n} w(t - ts_i)$$

$$LS^{(t)} = \sum_{i=1}^{n} w(t - ts_i) \cdot x_i$$

$$SS^{(t)} = \sum_{i=1}^{n} w(t - ts_i) \cdot x_i^2$$

donde n = es el número de objetos/datos, ts_i = al timestamp en el que se agrega el dato x_i al micro-cluster. Cabe destacar que la propiedad aditiva de CF se conserva. Ahora, si no se agrega ningún objeto a un CF durante el intervalo de tiempo [t, t + Δt], entonces:

$$CF^{(t+\Delta t)} = \omega(\Delta t) \cdot CF^{(t)}.$$

Los detalles sobre esta propiedad y la prueba correspondiente se pueden encontrar en [44].

El procedimiento de actualización de las entradas de los nodos se explica de la siguiente manera:

1. Cada dato de inserción x tiene un timestamp ts_x que indica el tiempo en el que el dato llegó al flujo.
2. Cada entrada en un nodo tiene un timestamp $e_s.ts$ que especifica su última actualización. Este último se utiliza, para calcular el tiempo transcurrido entre la última actualización y ts_x. Al descender en un nodo, actualizamos todas las entradas e_s de la siguiente forma:

$$e_s.CF \leftarrow \omega(t_x - e_s.t_s).e_s.CF$$
$$e_s.buffer \leftarrow w(t_x - e_s.t_s).e_s.buffer$$

3. El timestamp de cada entrada se restablece con el timestamp de x, por lo tanto $e_s.ts$ = ts_x.

Es importante aclarar aquí que todas las entradas en el mismo nodo obtienen el mismo timestamp global ya que todas las entradas se actualizan en el nodo al que desciende el punto de datos.

El procedimiento de ponderar los datos por tiempo ayuda a evitar divisiones y ahorrar tiempo. Pues, si un nodo q está a punto de dividirse, el algoritmo verifica si la entrada menos significativa i puede descartarse o no. Si este es el caso, se descarta i, dejando espacio para que se inserte el dato entrante, y evitando una división. Se deben actualizar todos los CF desde q hasta la raíz.

Manejo de flujos muy rápidos: aceleración a través de la agregación

El problema con las transmisiones rápidas es que las inserciones se interrumpirán continuamente en la raíz o en el nivel superior del árbol, y se agregarían una gran cantidad de elementos en sus buffers que tendrán pocas posibilidades de llegar a una hoja. Peor aún, datos diferentes que pertenecen a diferentes subárboles y hojas se fusionarán y se volverán inseparables en un buffer. Como conclusión, la calidad de los resultados se deteriorara si se producen interrupciones constantemente en niveles más altos.

Para resolver este problema, los autores proponen un speed-up a través de la agregación (buffers) antes de la inserción. Esto se logra al no insertar cada punto de datos individualmente, lo cual dejaría más tiempo para descender más profundo con los elementos en los buffers intermedios. Más bien, esto se logra sumando m puntos de datos entrantes e insertando el micro-clusters almacenado en el buffer, y no insertando cada elemento por separado.

El problema aquí es que los puntos de datos muy diferentes pueden ir al mismo buffer y obtenemos el escenario que se describió al principio de esta sección. Para resolver esto, los autores mantienen varios buffers para objetos diferentes y se aseguran de que los objetos resumidos en el mismo buffer sean similares. Para llevar acabo esto, se establece un radio máximo para la distancia máxima de un datos en un buffer. El valor o radio máximo se determina como la varianza promedio de las hojas. Por lo tanto, las transmisiones rápidas no deterioran la calidad del clúster desproporcionadamente.

Además, la cantidad de buffers a mantener está determinada por la velocidad de la secuencia, es decir, no puede exceder el número de cálculos de distancia que pueden realizarse entre dos elementos que llegan. En el caso de un flujo de datos variable, el usuario debe establecer la cantidad máxima de buffers, el cual, constituye el único parámetro de ClusTree.

Generación de Macro-Clusters

En este punto, podemos deducir que el resultado de agrupación producido por Clustree es el conjunto de micro-clusters (CF) que se encuentran almacenados en el nivel hoja del árbol, siendo este conjunto la mejor representación del flujo dependiendo de su velocidad. También podemos decir, que al proceso de armado del árbol lo podemos ver o considerar como el componente o fase online este algoritmo, lo cual permite utilizar cualquier enfoque de agrupamiento sobre estos datos en la fase offline. Entonces, tomando el centro de cada CF como representación de datos, se puede aplicar directamente algoritmos de agrupación como k-means, o incluso algoritmos basados en densidad para detectar conjuntos de datos de formas arbitrarias.

Una de las ventajas principales de este algoritmo es que puede mantener un número mayor de micro-cluster en comparación con otros enfoques como CluStream o DenStream, por lo tanto, la agrupación offline tiene una granularidad más fina con respectos a otros enfoques.

Resumen

Características importantes

- Mejora la calidad de clúster en transmisiones muy rápidas mediante el uso de buffers.
- La estructura jerárquica del árbol produce una complejidad de inserción logarítmica y permite tener micro-cluster con diferentes grados de granularidad.
- El uso de inserciones Anytime permite que Clustree haga una mejor utilización del tiempo de ejecución disponible y a partir de esto, brindar un modelo autoadaptativo dependiendo la velocidad de flujo.
- Genera micro-clusters actualizados en el tiempo mediante el uso del concepto de ‘Aging’ o envejecimiento de los elementos del flujo.
- Los micro-clusters pueden ser utilizado fácilmente para generar macro-cluster utilizando algoritmos populares de agrupación.

Limitaciones

- No resuelve de forma nativa el problema de detectar outlier.
- Complejidad alta de implementación, lo cual conlleva a que no sea fácilmente escalable en una arquitectura distribuida, ya que habría mucho overhead de información y comunicación entre los nodos para mantener actualizado de forma consistente cada buffer de cada nodo en el árbol.

CluStream

Dado que los datos de flujo imponen naturalmente restricciones de una única iteración en el diseño de los algoritmos, se hace más difícil proporcionar flexibilidad en el proceso de encontrar clusters utilizando algoritmos convencionales, y aún más difícil si se quiere generar agrupaciones en diferentes ventanas de tiempo, para tal caso requeriría un mantenimiento simultáneo de los resultados intermedios de los algoritmos de agrupamiento en las diferentes etapas. Tal carga computacional aumenta con la progresión del flujo de datos y puede convertirse rápidamente en un cuello de botella para la implementación en una metodología incremental. Además, en muchos casos, el usuario puede desear analizar los resultados de clustering en un instante previo y compararlos con los resultados actuales. Esto requiere una carga computacional aún mayor y puede volverse rápidamente difícil de manejar para flujos de datos rápidos.

Es por esto que CluStream [5] es uno de los algoritmos más populares en la literatura sobre agrupamiento debido a que es el primer algoritmo que define el enfoque online-offline sobre streams. Todo el proceso de clustering se divide en un componente de micro-clustering online que debe ser un proceso muy eficiente para la generación y almacenamiento de información resumida del flujo de datos y un componente de macro-clustering offline que utiliza los datos resumidos para proporcionar los resultados de clustering sobre el flujo cuando sea requerido por los usuarios. Este enfoque es ampliamente utilizado en muchos algoritmos de clustering de flujo debido a su eficiencia en el manejo de los flujos.

Micro-clusters:

La estructura de Micro-clustering resume la información del flujo de dato de forma eficiente y a la vez, almacena una referencia de la localización temporal de los datos, con el fin de facilitar la agrupación y el análisis en diferentes ventanas de tiempo. Definiendo a la acción de “resumir” como el proceso que se encarga de generar una representación o modelo para un subconjunto de datos dentro del flujo.

Definición de Micro clustering

El concepto de microclusters es similar al vector de características (cluster feature = CF) para un grupo que usa BIRCH[45] pero se agrega una extensión para almacenar información con respecto a la localidad temporal de los datos, por lo tanto, la definición formal de micro-cluster en CluStream es la siguiente:

Un micro-cluster para un conjunto de datos de d-dimensiones $X_{i1} \ldots X_{in}$ con timestamps $T_{i1} \ldots T_{in}$ es una (2 * d + 3)-tupla ($\overline{CF2^x}$, $\overline{CF1^x}$, $CF2^T$, $CF1^T$, n) . donde $\overline{CF2^x}$ y $\overline{CF1^x}$

corresponden cada uno a un vector de d entradas. La definición de cada una de las entradas de la tupla es es la siguiente:

- Para cada dimensión, la suma de los cuadrados de cada valor en los datos es guardado en $\overline{CF2^x}$. Por lo tanto, $\overline{CF2^x}$ es un vector con d valores y además podemos ver que es equivalente a la definición de SS [45] .
- Para cada dimensión, la suma de cada valor en los datos es guardado en $\overline{CF1^x}$. Por lo tanto, $\overline{CF1^x}$ es un vector con d valores y además podemos ver que es equivalente a la definición de LS[45] .
- La suma de los cuadrados de cada timestamp de dato $T_{i1} \ldots T_{in}$ es guardado en $CF2^T$.
- La suma de cada timestamp de dato $T_{i1} \ldots T_{in}$ es guardado en $CF1^T$.
- El número de datos que representa el micro-cluster está representado por n.

Los microclusters se almacenan y se agrupan dependiendo el momento en el que llegan al flujo, a estos grupos se los denomina **snapshots**. A la vez, los snapshots se almacenan siguiendo un patrón piramidal (**Pyramidal Time Frame**), donde este proporciona un trade-off efectivo entre los requisitos de almacenamiento y la capacidad de recuperar microclusters almacenados en diferentes momentos en el tiempo.

Pyramidal Time Frame:

En esta técnica, las snapshots se almacenan en diferentes niveles de granularidad dependiendo en el orden en el que llegaron. Las snapshots se clasifican en diferentes órdenes, que pueden variar de 1 a log(T), donde T es el tiempo de reloj transcurrido desde el comienzo de la transmisión de datos. El orden de una clase particular de snapshots define el nivel de granularidad de las snapshots. Las snapshots de diferente orden se mantienen de la siguiente manera:

- Las snapshots de order i-th se producen a intervalos de tiempo α^i, donde α es un entero mayor igual a 1 ($\alpha \geq 1$). Específicamente, cada snapshot de orden i-ésimo se toma cuando su timestamp de llegada (tiempo transcurrido desde el comienzo del flujo) es exactamente divisible por α^i.
- En cualquier momento dado en el tiempo, solo se almacenan las últimas $\alpha^l + 1$ snapshots i-ésimo orden. Esto se debe a que es imposible almacenar todos desde el comienzo de la transmisión de datos.

De la definición anterior podemos hacer las siguientes observaciones:

- Para un flujo de datos, el orden máximo de cualquier snapshot almacenada el tiempo T es igual a $log_a(T)$. Siendo T el tiempo transcurrido desde el comienzo del flujo.
- Para un flujo de datos, el número máximo de snapshot mantenidas en tiempo T es $(\alpha^l + 1).log_a(T)$.
- Para cualquier ventana de tiempo de tamaño h especificada por un usuario, al menos una snapshot puede ser encontrada en el rango de tiempo $(1 + 1 / \alpha^{l-1})$. La demostración de esta propiedad se puede ver en [5].

A parte de esto, de la definición anterior encontramos el siguiente problema, la técnica de Pyramidal Time Frame produce un nivel alto de redundancia en el contenido de las snapshots. Por ejemplo, dado el tiempo de reloj 8, es decir, T = 8, este es divisible por α^0, α^1, α^2 y α^3 cuando $\alpha = 2$. Por lo tanto, un microcluster que llega en tiempo T = 8 se almacenará en las snapshots tanto de orden 0, 1, 2 y 3, teniendo en este caso varias copias redundante del mismo microcluster.

Orden de Snapshots	Clock Times (Últimas 5 Snapshots)
0	55 54 53 52 51
1	54 52 50 48 46
2	52 48 44 40 36
3	48 40 32 24 16
4	48 32 16
5	32

Tabla 3.1: Un ejemplo de snapshots almacenada para $\alpha = 2$ y $l = 2$

Online Clustering con CluStream

La fase online del algoritmo no depende de ningún parámetro que el usuario deba especificar y tiene como objetivo mantener un resumen estadístico de los datos entrantes, para que puedan ser utilizadas de manera efectiva por la fase offline. En otras palabras, es la fase en la que se forman los microgrupos.

El algoritmo de esta fase funciona de manera iterativa, manteniendo siempre un total de q micro-clusters en cualquier momento de la ejecución, donde q está determinado por la cantidad de memoria principal disponible para almacenar micro-clusters.

Por lo tanto, q es significativamente más grande que el número natural de clústeres en los datos, pero también es significativamente más pequeño que el número de datos que llegan en un período de tiempo en la transmisión de un flujo infinito. Estos q microclusters forman

la snapshot actual que representa el estado del flujo con la llegada de los datos más recientes.

Al inicio de la ejecución del algoritmo, se almacenan en disco los primeros puntos P de la secuencia de datos y se ejecuta el algoritmo estándar de k-means para crear los q microclusters.

Después de la etapa de inicialización, se inicia el proceso de actualización de los microclusters creados. Entonces, cada vez que llega un dato nuevo al sistema, se produce alguna de las siguientes dos acciones:

1. El punto es absorbido por el microcluster existentes más próximo en función de alguna métrica de distancia entre el centroide del microcluster y el punto entrante. El microcluster más cercano se actualiza con el dato nuevo usando la propiedad aditiva de los microcluster mencionada anteriormente en este trabajo.

2. El punto se coloca en un microcluster, pero como siempre hay una cantidad q de microcluster, conlleva a que hay que liberar un espacio para almacenar el nuevo, lo cual se puede lograr eliminando uno de los microclusters más antiguos o fusionando 2 microclusters.

Si ocurre la segunda opción, donde un punto recibe su propio microcluster, primero se evalúa si se puede eliminar un microcluster, para esto se calcula el timestamp promedio de cada microcluster utilizando la información $CF1^T$ de timestamp almacenada en el microcluster. Entonces, si para un microcluster M, el timestamp promedio es menor que un umbral definido por el usuario, ese microcluster se elimina y se crea uno nuevo. En caso de que el timestamp promedio sea mayor para todo los microclusters disponibles, significa que son todos recientes, en tal caso, se fusionan los dos microclusters más cercanos.

Mientras que el proceso anterior de actualización se ejecuta cada vez que llega un dato nuevo, en esta fase también se ejecuta otro proceso adicional, el cual se inicia cada vez que tiempo de ejecución (clock time) es divisible por α^i para cualquier número entero i. En este proceso, el conjunto actual de micro-clusters se almacena en el disco como una nueva snapshot, y se indexa según su tiempo de almacenamiento (orden i). Las snapshots menos reciente de orden i se eliminan, si una cantidad $\alpha^l + 1$ de snapshots de dicha orden ya se han almacenado en el disco, y si además cada snapshot no es divisible por α^{i+1} ya que en este último caso, la snapshot continúa siendo una snapshot válida para orden (i + 1), en otras palabras, es una snapshot válida para el siguiente orden según el tiempo de ejecución. Estas snapshots se pueden usar para formar agrupaciones de mayor nivel o para realizar un análisis de la evolución del flujo de datos.

Offline Clustering con Clustream

En la fase offline de este algoritmo, la restricción de una sola pasada sobre los datos disponibles en el flujo no aplica, ya que se utiliza la información estadística que proporcionan los microclusters para generar los resultados de agrupación final. Cada microcluster se trata como pseudo puntos para el algoritmo de k-means (macro-clustering).

El usuario proporciona el número de clústeres k que se deben calcular y el intervalo o horizonte de tiempo h sobre el que se debe realizar la agrupación. Esta elección determina si el resultado formado es más preciso o no. El uso Pyramidal Time Frame es útil en este caso, ya que garantiza la disponibilidad de las snapshots que se deben usar para calcular los microclusters para el intervalo de tiempo definido por el usuario.

Características principales y limitaciones.

Características principales

- El enfoque de dos fases ayuda a capturar la información esencial rápida y continuamente en una sola pasada y al mismo tiempo generar buenos clusters usando algoritmos tradicionales.
- Brinda un nuevo enfoque de micro-cluster, el cual permite capturar la localidad temporal de los datos.
- Muy fácil de entender e implementar.
- Pyramidal Time Frame permite proporcionar una alternativa para evaluar la evolución de flujo en diferentes intervalos de tiempo.

Limitaciones

- Clustream es un algoritmo de clustering basado en particiones y, como cualquier algoritmo de esta clase, es sensible a valores atípicos.
- Al utilizar k-means, como algoritmo de macro-clustering siempre se obtiene un número fijo de grupos, por lo tanto, no genera un número dinámico de clusters.
- Los micro-clusters y los macro-clusters generados son siempre de forma esférica, es decir, no genera clusters de formas arbitrarias.
- Tratar con datos de alta dimensionalidad afecta la calidad de los resultados finales[46].

DenStream

Denstream, es un algoritmo de clustering basado en densidad, toma conceptos del algoritmo de Clustream y propone solucionar algunas de las restricciones que presentan los algoritmos de clustering basado en partición, como la suposición o predefinición del número de agrupaciones a formar y la generación de grupos con formas esféricas. Denstream presenta una técnica para descubrir un número dinámico de clusters y de formas arbitrarias sobre flujos que evolucionan en el tiempo.

Los autores de este algoritmo, describen que la limitación de memoria que impone naturalmente los flujo de datos es uno de los desafíos más difíciles a la hora de generar agrupaciones con formas arbitrarias debido a que este tipo de agrupaciones generalmente se representa utilizando todos los puntos en el grupo. Además de esto, agregan que no contar con información global del flujo es otro de los grandes desafíos, ya que los algoritmos convencionales basado en densidad necesitan de esta característica.

Denstream lleva a cabo el tratamiento del flujo utilizando el modelo de ventana damped windows (o también, llamado fadding windows), en la que el peso de cada punto de datos disminuye exponencialmente a medida que pasa el tiempo, lo cual se representa mediante una función de desvanecimiento o envejecimiento $f(t) = 2^{-\lambda \cdot t}$, donde $\lambda > 0$ y t representa un instante en el tiempo. La función de desvanecimiento se utiliza en aplicaciones en las que es deseable disminuir poco a poco la importancia de los datos más antiguos. Cuanto mayor sea el valor de λ, menor será la importancia (el peso) de los datos históricos en comparación con los datos más recientes.

Core-micro-cluster

Denstream utiliza una estructura de resumen similar a los micro-cluster de Clustream, pero esta técnica al trabajar con un modelo de ventana *damped,* la estructura del micro-cluster está ponderada por la importancia del tiempo y se denomina core-micro-cluster (o también c-micro-cluster). Su definición es la siguiente:

> Un core-micro-cluster en un determinado instante t y para un grupo de puntos cercanos $pi_1, ..., pi_n$ con timestamps $Ti_1, ..., Ti_n$ está definido como $CMC(w, c, r)$ donde:
>
> - $w = \sum_{j=1}^{n} f(t - Ti_j)$, es el peso (weigth) del core-micro-cluster,

- $c = \frac{\sum_{j=1}^{n} f(t-Ti_j)pi_j}{w}$ es el centro del core-micro-cluster,
- $r = \frac{\sum_{j=1}^{n} f(t-Ti_j)dist(pi_j, c)}{w}$ es el radio del core-micro-cluster, y $dist(pi_j, c)$ es la distancia euclidiana entre el punto pi_j y el centro c.

Además cada core-micro-cluster debe satisfacer que su peso debe ser superior o igual a μ, $(w \geq \mu)$ y el radio debe ser menor o igual a ε, $(r \leq \varepsilon)$, cuando se cumple estas condiciones se dice que el core-micro-cluster es un micro-cluster denso, siendo μ y ε parámetros de Denstream a definir por el usuario. La condición del radio sirve para indicar que la cantidad de c-micro-clusters es mucho mayor que la cantidad de clusters naturales. Y por otro lado, la cantidad de c-micro-clusters es significativamente más pequeño que el número total de datos en el flujo debido a su restricción del peso. Los core-micro-clusters serán utilizados en la etapa offline del algoritmo para generar la agrupación final.

En un flujo de datos en evolución, el rol de los clústeres y valores atípicos (outlier) a menudo se intercambia. Por lo tanto, Denstream para poder diferenciar cada c-micro-cluster entre estos roles establece las estructuras de "potencial core-micro-cluster" (p-micro-cluster) y "outlier-micro-cluster" (o-micro-cluster), la diferencia entre estas estructuras es que tienen distintas restricciones en cuanto al peso, en el caso de un p-micro-cluster se tiene que cumplir $(w \geq \beta\mu)$ y para los o-micro-cluster $(w < \beta\mu)$. Las definiciones formales de estas dos estructuras y a partir de la cual se deriva la implementación son las siguientes:

Un Potencial Core-micro-cluster en un instante de tiempo t para un grupo de puntos o datos cercanos $pi_1, ..., pi_n$ con sus respectivos timestamps $Ti_1, ..., Ti_n$, se define como una 3-tupla $\{ \overline{CF^1}, \overline{CF^2}, w \}$ donde:

- $w = \sum_{j=1}^{n} f(t - Ti_j)$, representa el peso y además se cumple $(w \geq \beta\mu)$ y $(0 < \beta \leq 1)$.
- $\overline{CF^1} = \sum_{j=1}^{n} f(t - Ti_j)\, pi_j$, representa la suma lineal ponderada o pesada del grupo de datos.
- $\overline{CF^2} = \sum_{j=1}^{n} f(t - Ti_j)\, pi_j^{\,2}$, representa la suma cuadrática ponderada o pesada del grupo de datos.
- Siendo β el parámetro que determina el límite para diferenciar entre p-micro-cluster y o-micro-cluster

Manteniendo estos valores es posible calcular las siguientes propiedades computadas: el centro del p-micro-cluster, $c = \frac{\overline{CF^1}}{w}$ y el radio $r = \sqrt{\frac{\overline{CF^2}}{w} - (\frac{\overline{CF^1}}{w})^2}$ con $(r \leq \varepsilon)$.

> Un Outlier micro-cluster en un instante de tiempo t para un grupo de puntos o datos cercanos $pi_1, ..., pi_n$ con sus respectivos timestamps $Ti_1, ..., Ti_n$, se define como una 4-tupla $\{ \overline{CF^1}, \overline{CF^2}, w, t_o \}$ donde: las definiciones de $\overline{CF^1}, \overline{CF^2}, w$, radio y centro son las mismas a las de p-micro-cluster. $t_o = Ti_1$ y representa el tiempo en el que fue creado el o-micro-cluster. Además se tiene que satisfacer que $(w < \beta\mu)$ lo cual determina que este micro-cluster es un outlier.

Por último, unas de las características más importantes de los p-micro-cluster y los o-micro-cluster es que cumplen las mismas propiedades que tienen los cluster feature (cf) de Clustream pero ajustándose a las características de peso. Entonces los p-micro-cluster y los o-micro-cluster poseen las propiedad de ser mantenidos incrementalmente (propiedad 3.1 en[14]), por ejemplo, sea un p-micro-cluster $c_p = (\overline{CF^1}, \overline{CF^2}, w)$, si no se fusiona ningún punto a c_p en un intervalo de tiempo δt c_p será actualizado de la siguiente forma:

$$c_p = (2^{-\lambda\delta t} \cdot \overline{CF^1}, 2^{-\lambda\delta t} \cdot \overline{CF^2}, 2^{-\lambda\delta t} \cdot w)$$

Y si un punto p necesita fusionarse con c_p en el instante actual, c_p se actualizará como:

$$c_p = (\overline{CF^1} + p^2, \overline{CF^2} + p, w + 1)$$

en este último caso, el peso se incrementa en uno (1) ya que el valor de la función de envejecimiento $f(t - Ti)$ da como resultado 1.

Demostración:
Sea t el instante de tiempo actual, en el cual se desea agregar punto p_i a c_p, entonces el timestamp de p_i, Ti, va ser igual a t, $(t = Ti)$. Por lo tanto:

$$f(t - Ti) = f(t - t) = f(0) = 2^{-\lambda \cdot 0} = 2^0 = 1$$

dando como resultado que el peso de p_i es $w = 1$, $\overline{CF^1} = 1 \cdot p_i$ y $\overline{CF^2} = 1 \cdot p_i^2$

Metodología de procesamiento:

El proceso completo de clustering sobre los datos del flujo se divide en 2 partes al igual que Clustream:

1. La fase online que se encarga de mantener los micro-cluster, en particular de mantener los p-micro-cluster y los o-micro-cluster.
2. La fase offline que se encarga de generar los macro-cluster o clusters final usando los p-micro-cluster de la fase online.

Fase Online: Micro-clusters

En esta etapa, es la que se encarga de generar los core-micro-cluster, específicamente se mantiene en memoria un grupo de p-micro-cluster a partir de los datos recientes del flujo, y por otro lado, los o-micro-cluster se mantienen en un espacio de memoria separado llamado buffer de outliers.

Cuando un dato o punto nuevo p llega al flujo, el proceso de agrupación de los core-micro-cluster es el siguiente:

1. Primero, se intenta unir el punto p con su p-micro-cluster c_p más cercano. Si r_p, el radio nuevo de c_p es menor o igual que ε, entonces se produce la fusión p con c_p. Esto se logra, utilizando la propiedad incremental descripta más arriba.

2. En caso que no se pueda realizar la fusión de p con su p-micro-cluster más cercano, se intenta unir p con su o-micro-cluster más cercano c_o. Si r_o, el radio nuevo de c_o, es menor o igual que ε, entonces se produce la fusión p con c_o. Ahora si w es mayor a $\beta\mu$, significa que c_o se puede convertir en un p-micro-cluster, por definición de potencial core-micro-cluster, entonces se eliminar c_o del buffer de outlier y se crea un nuevo p-micro-cluster a partir de c_o.

3. Si no se puede realizar ninguna de las dos anteriores acciones, entonces se crea un nuevo o-micro-cluster c_o con los valores de p y se agrega en el buffer de outliers. En otras palabras, p debe ser un outlier.

Al mismo tiempo, que ocurre el proceso de inserción de datos, para cada p-micro-cluster c_p que no se haya fusionado con ningún dato nuevo, se actualiza disminuyendo gradualmente su peso w de la forma que lo establece la propiedad 3.1[14]. Luego de esto, si $(w < \beta\mu)$, significa que c_p pasa a ser un o-micro-cluster, por lo que se debe eliminar c_p de la memoria y almacenarlo en el buffer de outlier. Este proceso de chequeo se realiza cada T_p periodo

de tiempo, lo que asegura, que no se realizan chequeo tan frecuentes y por lo tanto, no se haga un mal uso del tiempo de procesamiento. T_p se define de la siguiente forma:

$$T_p = \frac{1}{\lambda} log(\frac{\beta\mu}{\beta\mu - 1})$$

En caso de que haya mucho ruido en los datos, la cantidad de o-micro-clusters puede aumentar rápidamente y mantenerlos todos en el buffer de outliers es costoso, por lo que DenStream soluciona este problema eliminando los o-micro-clusters cuyo peso en el instante t_c actual es menor que su límite inferior de peso ξ, que se define de la siguiente forma:

$$\xi(t_c, t_o) = \frac{2^{-\lambda(t_c - t_o + T_p)} - 1}{2^{-\lambda T_p} - 1}$$

donde t_o es el tiempo de creación del o-micro-cluster.

Los autores prueban matemáticamente (Teorema 4.1 de [14]) que el número total de micro-clusters aumenta logarítmicamente con el paso del tiempo y, al mismo tiempo, afirman que el número total de microclusters en aplicaciones reales no va a ser muy grande. La prueba está fuera del alcance de esta explicación.

Inicialización de los core-micro-clusters

Para inicializar la fase online, Denstream ejecuta DBSCAN [35] sobre los primeros datos $\{P\}$, los autores no aclaran cuál es la métrica o la cantidad de datos a utilizar para inicializar.

Por lo tanto, para generar el grupo inicial de p-micro-clusters, se itera por cada punto p en $\{P\}$, y si el peso total (en este caso, el número de puntos) en su vecindad ε es mayor o igual a $\beta\mu$, se crea un p-micro-cluster con p y sus vecinos, y luego se eliminan de $\{P\}$.

Fase Offline: Generación de resultados

El conjunto de p-micro-clusters obtenidos en la fase online, son utilizados para aplicar una variante del algoritmo DBSCAN para obtener el resultado final de agrupacion (macro-clusters), cada p-micro-cluster c_p es considerado como un punto virtual (virtual point) ubicado en el centro c_p y con peso w.

De forma similar al algoritmo DBSCAN original[35], la variante crea macro clusters basado en la cercanía relativa de los p-micro-clusters mediante el concepto de *density-connected* [35], es decir, todos los p-micro-clusters densamente conectados (*density-connected*)

forman un clúster, pero adaptando este concepto con los parámetros de peso y distancia μ y ε:

Directly density-reachable: un p-micro-cluster c_p es directly density-reachable o directamente alcanzable desde un p-micro-cluster c_q con parámetros ε y μ, si el peso w de c_q es mayor que μ, $(w > \mu)$ y $dist(c_p, c_q) \leq r_p + r_q$, donde $dist(c_p, c_q)$ es la distancia entre los centro de c_p y c_q.

Density-reachable: un p-micro-cluster c_p es density-reachable o alcanzable desde un p-micro-cluster c_q con parámetros ε y μ, si existe una cadena de p-micro-clusters $C_{p1}, ..., C_{pn}$, $C_{p1} = c_q$, $C_{pn} = c_p$ tal que C_{pi+1} es directamente alcanzable (directly density-reachable) desde C_{pi}.

Density-connected: un p-micro-cluster c_p es density-connected o está densamente conectado a un p-micro-cluster c_q con parámetros ε y μ,si existe un p-micro-cluster c_m tal que ambos c_p y c_q son alcanzables (density-reachable) desde r c_m con parámetros ε y μ.

Características importantes y limitaciones

Características importantes

- Define una estrategia para diferenciar ruido (noise o outliers) y p-micro-cluster.
- Muy útil para flujos de datos que evolucionan dinámicamente y para detectar concept-drift.
- Generar clusters de formas arbitrarias.

Limitaciones

- Aunque se eliminen periódicamente algunos o-micro-clusters, el número de microclusters puede aumentar mucho y exceder las limitaciones de memoria.

Capítulo 4

D3CAS: Nuevo algoritmo para Streaming Clustering

Análisis y Motivación

Tras analizar y estudiar los trabajos más importantes relacionados con la detección de clusters y los flujos de datos los cuales pueden ser evolutivos o infinitos se ha llegado a la deducción que un algoritmo ideal de clustering debe cumplir con los siguientes requerimientos:

1. Procesamiento en una sola iteración: Esta es una restricción natural que tienen las secuencias de datos debido a que pueden tener un volumen de datos muy grande, tener una gran velocidad o ser infinitos, entonces realizar varias iteraciones sobre el mismo conjunto de datos produciría que los datos posteriores sean descartados sin ser procesados.

2. Bajo uso de memoria y CPU: El sistema debería poder procesar flujos muy grandes o infinitos utilizando sólo la memoria principal, la cual tiene un tamaño mucho más pequeño que la del flujo.

3. No conocer a priori ni número ni formas de las agrupaciones: En la agrupación, el usuario a menudo sabe muy poco acerca de cómo los datos se agrupan antes del procesamiento. Además, en los flujos, la distribución de los datos puede cambiar en el tiempo, lo que implica que las agrupaciones también cambien, ergo, el sistema debe ser capaz de detectar estos cambios.

4. La capacidad para filtrar el ruido en flujos que evolucionan continuamente. El ruido aleatorio puede ocurrir en cualquier parte de la secuencia y filtrarlo puede ayudar a obtener un buen resultado de agrupamiento.

5. Descubrir agrupaciones en ventanas de datos. Los flujos siempre están evolucionando a lo largo del tiempo y el algoritmo de agrupamiento debe mantener un número limitado de agrupaciones para los datos más relevantes o recientes (ventana) para restringir el uso de la memoria pues es imposible mantener todas las agrupaciones (resultados) para cada instante del flujo.

6. Representación compacta de los datos: No es posible almacenar toda la secuencia de datos para representar las agrupaciones como se realiza en los métodos de agrupamiento tradicionales sobre un conjunto estáticos de datos, ya que el volumen del flujo puede ser enorme o infinito. Por lo tanto, se necesita una representación compacta y limitada en memoria para las agrupaciones a formar, que no sólo tenga la capacidad de mostrar el estado actual de la secuencia de datos a lo largo del tiempo sino que también utilice el menor espacio posible en memoria.

7. La habilidad de mantener agrupaciones para distintos puntos en el tiempo: Aunque ya se mencionó que es imposible mantener todos los resultados para cada instante en el tiempo, se tendrá que contemplar la opción de mantener resultados para algunos puntos en el tiempo (snapshot), lo cual trae el desafío de buscar un balance entre el espacio necesario para mantener las representaciones de los datos actuales y el espacio para las snapshots.

8. Tener la capacidad de detectar agrupaciones sin utilizar parámetros asignados por el usuario: Lograr una técnica capaz de detectar agrupaciones sin que el usuario especifique parámetros como la cantidad de agrupaciones a detectar o cantidad de elementos a procesar, se lo puede traducir como un sistema que es fácil de utilizar, pero esto implica tener que optimizar continuamente y automáticamente estos parámetros, lo cual se traduce como un aumento de procesamiento.

9. Procesar sobre una arquitectura distribuida (escalabilidad): Con la existencias de flujos de datos de gran volumen y dimensionalidad, hay casos en los que es imposible procesarlo utilizando una sola computadora, por lo cual, se necesita implementar técnicas que sean capaces de correr sobre una arquitectura distribuida lo que trae como dificultad que no se puedan utilizar las técnicas tradicionales de clustering ya que no están implementadas o pensadas para trabajar en un ambiente distribuido.

Como bien se dijo al principio de esta sección, si se cumplen todas las características se tendría un algoritmo ideal de detección de clustering, lamentablemente ninguno de los algoritmos de agrupamiento sobre flujos de datos existentes puede resolver todos los requerimientos mencionados, por lo tal motivo, cada técnica de clustering se especializa en resolver algunos de los desafíos presentados. Es por esto, que la elección del algoritmo depende en gran medida del caso de uso o estudio.

El caso de estudio e investigación de esta tesina está enfocado en el diseño de un técnica de detección de clusters que sea dinámica, específicamente en la detección dinámica en el número de grupos a detectar, en consecuencia, el diseño prioriza los requerimientos relacionado con este caso de uso, como por ejemplo la detección de grupos con formas arbitrarias, pero sin dejar de lados lo desafíos naturales que traen los flujos de datos como el procesamiento en una sola iteración, bajo uso de la memoria principal, contemplar el cambio en la distribución de los datos y procesar en tiempo real. En la siguiente sección se empieza a definir el diseño y las justificaciones de las decisiones tomadas.

Diseño

En esta sección se presentan las características del algoritmo de esta tesina, el cual recibe el nombre de **D3CAS**: **"Distributed Dynamic Density based Clustering Algorithm for dataStream"**. Este algoritmo está basado principalmente en las estructuras de datos y metodología de procesamiento tanto del algoritmo de CluStream y DenStream.

El desafío de D3CAS, recae en lograr un algoritmo que funcione en una arquitectura distribuida, específicamente, utilizando el modelo de Apache Spark Streaming, el cual, como se mencionó anteriormente (Capítulo 2), al igual que MapReduce, se basa en la arquitectura Master-Worker, lo cual representa la característica diferencial más importante con respecto a las técnicas mencionadas en los capítulos anteriores, ya que todas están diseñadas para trabajar en un ambiente no distribuido.

El framework Apache Spark, junto con Hadoop vienen siendo los modelo más utilizado en la actualidad para el procesamiento de Big Data y Flujos de datos, siendo los framework más populares en esta área ya que tienen un gran peso tanto en los ambientes académicos como en la industria de software.

Ventana de Tiempo

Para realizar el tratamiento del flujo de dato, se decide utilizar el modelo de ventana de tiempo **Fadding o Damped windows**, el cual es utilizado, por ejemplo, por DenStream. Se tomó esta decisión debido a la ventaja que presenta este modelo, la cual ya se explicó tanto en el capítulo 1 (sección: “Ventanas de tiempo”) como en la explicación de DenStream (capítulo 3, sección “DenStream”), este tipo de ventana sirve para darle importancia a los datos más recientes, lo cual facilita la detección temprana de cambios en la distribución de los datos.

En este modelo, los datos deben almacenar un atributo que determine su importancia en el tiempo. Dicho valor lleva el nombre de *peso* y es calculado por medio de una función de desvanecimiento o envejecimiento, en la que el peso de cada dato disminuye exponencialmente a medida que pasa el tiempo, en otras palabras, cuanto más chico sea el resultado de la función, tiene menos importancia en el tiempo. La función elegida, es similar a la definida en DenStream:

$$f(dt) = 2^{-\lambda \cdot dt}$$

donde $\lambda > 0$ y dt representa la diferencia entre timestamp de llegada del dato y un instante de tiempo mayor al de llegada, generalmente representado por el instante de tiempo actual. La diferencia entre esta función y la utilizada en DenStream, es que en DenStream, en vez de usar dt, se utiliza t, donde t, se representa como un instante de tiempo diferente del timestamp del dato, lo cual es confuso, pues cuando los autores de DenStream definen la propiedad peso w de un micro-cluster, la función de desvanecimiento no toma una referencia de tiempo, sino que toma la duración de un intervalo de tiempo, lo que justamente se define en esta tesina como dt.

Una característica importante a destacar es que cuanto mayor sea el valor de λ, menor será la importancia (el peso) de los datos en comparación con los datos más recientes. En esta implementación debido a la importancia de la variable λ, se la considera como una variable pública, la cual el usuario la puede modificar dependiendo del contexto del flujo que se esté por analizar.

Metodología Online-offline

Esta metodología o enfoque es utilizando tanto en CluStream, ClusTree y DenStream, siendo CluStream el primer algoritmo en implementarlo. Este enfoque separa la ejecución en dos etapas, una online, que consume los datos del flujo y realiza un procesamiento rápido sobre estos para evitar que se descarten datos debido a la velocidad de llegada de los datos. Luego, el resultado de este procesamiento es almacenado en la memoria principal para que la etapa offline los utilice para realizar un procesamiento o análisis más intensivo en un momento determinado en el tiempo o cuando el usuario lo especifique.

En las tareas de clustering sobre flujos de datos, la ventaja de usar esta metodología radica en que mientras se generan micro-clusters en la etapa online paralelamente en la etapa offline, se puede ejecutar una adaptación de un algoritmo tradicional de clustering sobre los micro-clusters generados con el objetivo de detectar agrupaciones finales. Lógicamente, el proceso de detección de agrupaciones se hace en la etapa offline debido a que los algoritmos tradicionales realizan un procesamiento más exhaustivo de la información ya que realizan múltiples lecturas sobre el mismo conjunto de datos.

Online-offline sobre la arquitectura distribuida

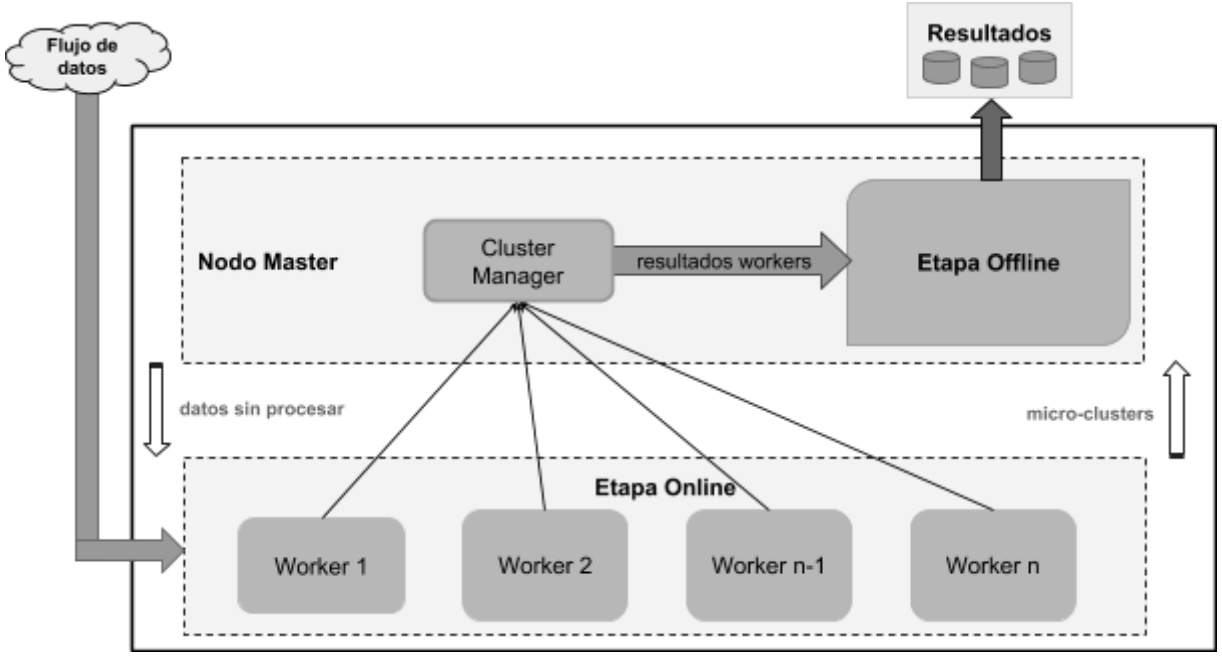


Figura 4.1: Diseño Online-Offline sobre Spark

El modelo de Spark, técnicamente trabaja sobre una arquitectura distribuida utilizando el patrón Master-Worker, donde las tareas o operaciones de *transformación* (Transformations), las cuales se encargan de transformar el tipo de los datos de entrada a otro tipo o dominio

diferente, se ejecutan paralelamente en cada uno de los nodos Worker. Mientras que el nodo Master se encarga de las siguientes dos funciones: primero, asigna las tareas de *transformación* junto a una partición de los datos de entrada a los nodos workers libres; segundo recolecta los resultados generados por los workers y una vez recolectado todos los resultados de cada worker, lleva a cabo el procesamiento o análisis correspondiente a los fines del sistema o aplicación, en el caso de esta tesina, dicho procesamiento es la detección de Agrupaciones.

Como se propone utilizar tanto el modelo de Spark como la metodología online-offline, surge el desafío de diseñar la lógica de la metodología sobre el modelo, respetando además, el patrón Master Worker. Como consecuencia se establece el siguiente diseño (figura 3.1), el cual será utilizado en este trabajo:

- La etapa o fase Online se ejecuta paralelamente en los workers debido a que la tarea de generar los micro-clusters se ajusta al tipo de procesamiento que realizan las operaciones de *Transformación*, ya que se encarga de procesar los datos de entrada para transformarlos en micro-clusters (es decir, realizan una transformación del dominio de los datos de entrada).

- La etapa o fase Offline, se ejecuta en el nodo Master, ya que para llevar a cabo la tarea de detección de agrupaciones necesita conocer todos los micro-clusters generados, dicho de otra manera, la tarea de clustering trabaja ajustándose al modelo de una operación de *Accion* (action) de Spark, ya que necesita primero juntar todos los micro-clusters y luego generar un resultado a partir de estos.

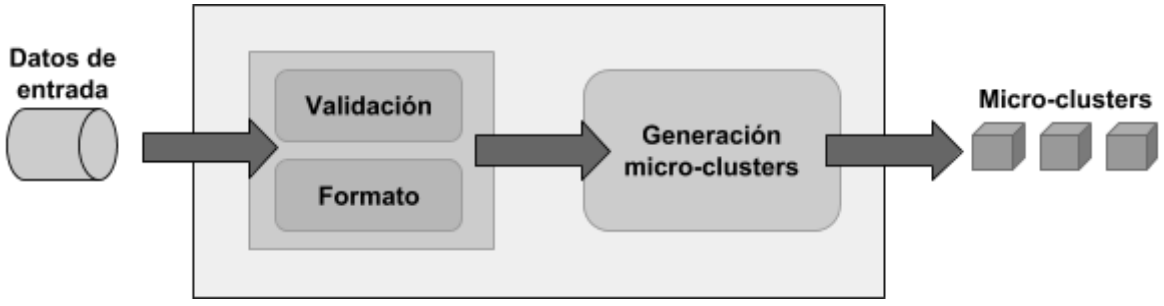


Figura 4.2: Procesamiento en la fase Online

Formato de entrada

Como no existe un estándar en el formato de los flujos de datos, pueden existir mucha diferencias entre la estructuración de la información entre un flujo y otro, por ejemplo, pueden existir flujos que sean transmitidos bajo el formato ARFF, JSON, o bajo el formato CSV (en estos casos separados por ‘,’ o ‘;’) o peor aún que no tengan un formato estandarizado, en estos casos considerados como datos crudos. Por tal motivo, se toma como primer instancia, llevar a cabo la tarea de analizar sintácticamente los datos de entrada con el objetivo de convertir la información proveniente en una estructura genérica adecuada para llevar a cabo el procesamiento de clustering.

En el contexto de la detección de agrupaciones, la estructura básica de procesamiento se denomina como punto (*point*) o ejemplo (*example*), y se estructura como un vector o arreglo de d dimensiones, donde cada dimensión representa un atributo o propiedad de la información. Otra característica importante dentro de este contexto, es que la gran mayoría de los datos a ser analizados por las tareas de clustering son de tipo numérico. Por tal característica, en este tesina, se aceptan sólo como flujos válidos, a aquellos flujos donde todos sus atributos sean de tipo numérico.

Micro Clusters

Esta técnica es implementada por primera vez en el algoritmo BIRCH, y es utilizado tanto en CluStream y DenStream. Consiste en reducir el volumen de datos de un conjunto o de una ventana del flujo en un modelo más pequeño y representativo, el cual puede ser almacenado en memoria principal. A parte del ahorro del espacio, esta técnica debe ser un proceso liviano y rápido, para evitar la restricción en la que los datos no son procesados en los flujos de gran volumen y velocidad.

Modelo Micro-cluster

Al utilizar el modelo de ventana damped windows, el diseño de los micro-clusters deben contener una propiedad o atributo para representar la importancia del mismo en el tiempo, este atributo comúnmente recibe el nombre de peso o weight. Por motivos de este requerimiento, el modelo de micro-clusters está basado y es análogo a la estructura *potencial core micro cluster (p-micro-cluster)* de la técnica Denstream, pero se agregan algunos atributos extras a fin de tener más información acerca del micro-cluster y para facilitar el entendimiento y cálculo de las propiedades computadas que se pueden estimar por medio de esta estructura. El modelo de micro-clusters es el siguiente:

Sea f la función de envejecimiento definida en el modelo de ventana, un micro-cluster en un

instante de tiempo t para un conjunto P de n puntos o datos cercanos de d dimensiones $p_1, ..., p_n$ con sus respectivos timestamps $T_1, ..., T_n$, se define como una 7-tupla $MC = \{ LS, SS, w, n, d, t_c, t_m \}$ donde:

- n representa la cantidad de puntos cercanos en el conjunto P.
- d representa la dimensionalidad de los puntos en el conjunto P.
- t_c representa el timestamp del instante de tiempo cuando fue creado el micro-cluster.
- t_m representa el timestamp de la última actualización del micro-clusters.
- $w = \sum_{j=1}^{n} f(t - Ti_j)$, representa el peso o importancia del micro-cluster en el tiempo.
- $LS = \sum_{i=1}^{n} f(t - T_i)\, p_i$, representa la suma lineal ponderada o pesada del grupo de datos, es decir, la suma lineal influenciada por la importancia en el tiempo. En la etapa de generación de micro-clusters, los datos se representan como puntos los cuales a la vez se representan como vectores de d dimensiones (debido a que primero pasan por la etapa de validación y formato), entonces la suma lineal (LS), según el álgebra vectorial, en simples palabras se define como la sumatoria entre vectores de d dimensiones.
- $SS = \sum_{j=1}^{n} f(t - Ti_j)\, pi_j^{\,2}$, representa la suma cuadrática ponderada o pesada del grupo de datos, es decir, la suma cuadrática influenciada por la importancia en el tiempo. En este caso la suma cuadrática se define como la sumatoria entre vectores de d dimensiones, donde los datos de cada dimensión se encuentran elevados al cuadrado.

Teniendo estos atributos se pueden computar las siguientes propiedades computadas:

$$centro = \frac{LS}{w} \text{ y } radio = \sqrt{\frac{SS}{w} - \left(\frac{LS}{w}\right)^2}$$

No sólo estamos tomando características de los p-micro-cluster, al mantener un atributo para representar la cantidad de elementos en el micro-clusters, también estamos tomando características directamente de los CF, cluster feature de BIRCH y CluStream. A consecuencia, el modelo de micro-clusters planteado hereda la propiedad de adición demostrada por medio de la álgebra vectorial en la reseña de BIRCH del capítulo 3. De acuerdo con el teorema de adición de CF, se puede actualizar incremental y consistentemente a medida que se fusionan micro-clusters o se insertan nuevos puntos de datos.

Generación

Dado a las características del modelo de Spark, el proceso de micro-clusters corre sobre los nodos workers en la etapa Map o Transformaciones del modelo, donde cada nodo poseen una partición de los datos actuales del flujo. Los datos en una partición son procesados conjuntamente para la producción de micro-clusters, a diferencia de las técnicas mencionadas en los capítulos anteriores donde se procesan los datos secuencialmente uno a uno.

Otra diferencia entre este proceso y el de las otras técnicas, es que el proceso de esta tesina solamente se encarga de realizar la formación de micro-clusters en cambio, en las otras técnicas, llevan a cabo tanto el proceso de formación y de actualización, lo cual agrega otra dificultad o restricción a esta tarea, como la necesidad que los datos del flujo y los micro-clusters deben estar disponibles globalmente, por lo que llevar a cabo el proceso de formación y de actualización bajo el modelo de Spark se vuelve poco escalable e ineficiente.

El proceso de generación está basado en la propiedad de distancia entre puntos y consiste en reducir la cantidad de datos en áreas de radio e. La generación se lleva a cabo de la siguiente forma: Para todos los puntos p_i de la partición de datos P_k, se buscan todos los puntos q_j cuya distancia euclidiana, $dist(p_i, q_j)$, sea menor a e, es decir, $dist(p, q_i) < e$, siendo e un parámetro global de la técnica a especificar por el usuario. Luego del filtrado se pueden tener los siguientes casos:

- Que no se haya encontrado ningún punto q_i, entonces en este caso, se crea un micro-clusters únicamente para representar el punto p_i: $MC(p_i, p_i^2, t_i, 1, 1)$ y luego se elimina p_i de la partición P_k.

- Que se haya encontrado un conjunto de puntos cercanos a p_i a una distancia menor a e, entonces en este caso, sea el conjunto Q que se encuentra formado por p_i y por el conjunto de puntos cercanos a p_i, se crea un micro-clusters de la siguiente forma: $MC(LS(Q), SS(Q), tq, cant(Q), cant(Q))$, y luego se eliminan todos los elementos de Q de la partición P_k.

Para los escenarios en los que ocurre el primer caso, se realiza la siguiente justificación, es necesario crear un micro-cluster que represente al punto p_i, ya que al no contar con la información global de todos los datos entrantes del flujo, no se lo puede considerar como ruido o outlier debido a que podría formar parte de un área dentro de otra partición, y por ende no se lo puede descartar. Y en caso de que efectivamente exista un micro-cluster en otra partición que contenga a p_i, tener dos micro-clusters que representan el mismo área o que sean muy similares, no afectará al algoritmo de clustering de la etapa offline, ya que

este último al estar basado en un algoritmo de densidad, no perjudican a las propiedades de densidad sino que las fortalecen.

Luego de este proceso, los nodos workers, se encargan de enviar el conjunto de micro-clusters generados al nodo master, quien será el encargado de realizar el proceso de actualización y llevar a cabo el proceso de detección de clusters.

Offline

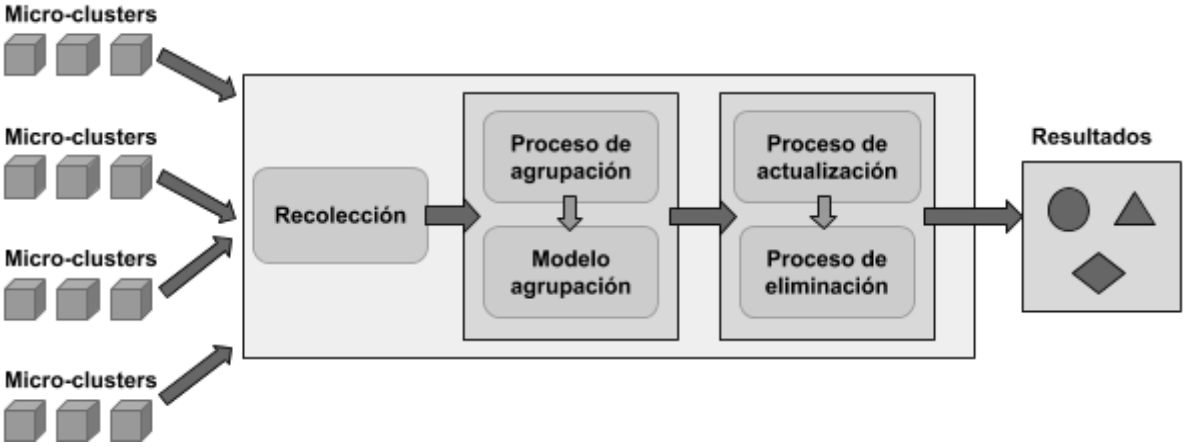


Figura 4.3: Procesamiento en la fase offline.

Recolección

La etapa de recolección es la más sencilla de todo el proceso, debido a que se encarga de juntar todos los micro-clusters generados en la etapa Online por los nodos workers. De esta forma, al terminar este proceso, se tienen todos los micro-clusters que representan todo el flujo de datos en un instante dado.

Agrupación basa en densidad

Para cumplir con los requerimientos del caso de estudio planteado, en la etapa offline de en esta implementación, el proceso de clustering se llevará a cabo mediante la utilización de una técnica basada en densidad, particularmente DBSCAN, la cual permite detectar agrupaciones sin asumir el número de agrupaciones a formar, es decir, detectar agrupaciones dinámicamente sin conocer de antemano la cantidad exacta de agrupaciones que pueden existir en el conjunto de datos. A parte de esto, DBSCAN, también proporciona la posibilidad de detectar agrupaciones con formas arbitrarias y detectar ruido o outlier presente en los datos. Como desventaja, al igual que cualquier método basado en densidad, se generan agrupaciones de baja calidad cuando la densidad entre diferentes agrupaciones son diferentes.

Para este proceso, se implementó el algoritmo original de DBSCAN y se logró incorporarlo como el método encargado de generar las agrupaciones finales sin modificar o adaptar la técnica al modelo de micro-cluster utilizado, para lograrlo, se tomó la decisión de utilizar los micro-clusters generados como puntos virtuales representados por el centro de cada micro-cluster, que a diferencia del algoritmo de DenStream, este último utiliza una versión modificada de DBSCAN adaptando el algoritmo original a las propiedades del modelo de microcluster generado, lo cual implica un mayor dificultad de implementación.

Como conclusión, el diseño de micro-clusters planteando permite utilizar la técnica original de DBSCAN obteniendo buenos resultados sin tener que adaptar ni el método de clustering ni el diseño de los micro-clusters. Si generalizamos esto último, podemos llegar a la deducción, que por medio de la representación de punto virtual planteada, en la etapa offline se puede utilizar cualquier algoritmo de clustering tradicional, es decir, sin realizar una adaptación al modelo. Sin embargo, siempre hay que considerar que lo recomendable es utilizar una técnica que no utilice memoria ni cpu en exceso, debido a las restricciones que imponen los flujos de datos, y más aún cuando el flujo tiene un gran volumen de información.

Actualización temporal

En esta etapa se lleva a cabo el proceso de actualización del peso de los micro-clusters acorde con el modelo de ventana de tiempo damped windows. A diferencia de CluStream y DenStream, que la actualización del peso de los datos se realiza en la etapa online, en este trabajo se realiza en la fase offline, debido a que recién en esta etapa se tienen todos los micro-clusters del flujo, ya que los workers, encargado de llevar a cabo de fase online, solo tienen sus respectivos micro-clusters generado a partir de los datos recién llegado en el flujo.

La actualización de los micro-clusters se realiza periódicamente, y específicamente se actualizan los atributos de peso w y los atributos $LS\ y\ SS$ que también están influenciados por la importancia del tiempo. La actualización de peso, se realiza utilizando la función de enjevecimiento definida por el modelo de ventana de tiempo $f(dt) = 2^{-\lambda \cdot dt}$, y el detalle del proceso de actualización se muestra a continuación.

Sea t el instante de tiempo actual, t_m el timestamp de última modificación del micro-cluster y dt la diferencia entre t y t_m, es decir, $t - t_m$, entonces cada micro-cluster actualiza sus atributos de la siguiente manera:

$$w = 2^{-\lambda \delta t} \cdot w$$
$$LS = 2^{-\lambda \delta t} \cdot LS$$
$$SS = 2^{-\lambda \delta t} \cdot SS$$

La función de envejecimiento es una función exponencial decreciente, entonces el resultado de la función siempre será un valor menor a 1, entonces al multiplicarlo por los atributos actuales del micro-clusters, hará que el peso del micro-cluster vaya disminuyendo gradualmente.

Luego del proceso de actualización del peso, se lleva a cabo el proceso de eliminación de los micro-clusters que se consideran vencidos o viejos según su peso. Un micro-cluster se considera viejo y debe ser descartado si su peso w es menor al umbral μ, es decir, $w < \mu$, siendo $0 < \mu \leq 1$. El umbral μ es un parámetro global de esta técnica, por lo que el usuario es el responsable de asignar un valor adecuado dependiendo del contexto del problema que se esté por analizar. Si μ es un valor muy cercano a 1, entonces sólo se conservarán los datos más recientes en el flujo, y cuando sea más cercano a 0, más grande será la ventana de tiempo representada por los micro-clusters.

Hay que determinar el periodo de tiempo en el cual se actualizan los datos tanto para el objetivo de descartar los que se consideran más viejo como para liberar memoria. Este periodo de tiempo no puede ser muy frecuente, ya que implicaría que se está realizando un mal uso de los recursos de procesamiento, y tampoco muy esporádico ya que se utilizaría demasiado espacio de memoria. Para solucionar este problema se decide utilizar la métrica de tiempo T_p expuesta por DenStream, la cual es utilizada por otras técnicas de clustering y además está demostrada matemáticamente por los autores de esta técnica: $T_p = \frac{1}{\lambda} log(\frac{\mu}{\mu - 1})$, entonces el proceso de actualización de los micro-clusters se lleva cada T_p instante de tiempo.

Modelo de los resultados

Dado que los datos de entrada provienen de un stream de datos, y con cada lote que llega los resultados producidos por el algoritmo pueden cambiar, resulta práctico que los resultados del proceso de clustering forme también un stream.

El stream de resultados o salida, está conformado por una estructura que contiene 2 objetos, donde uno de los objetos representa los clusters detectados junto a los centros de los micro-clusters que los conforman y el otro objeto representan los puntos que son considerado ruidos o outliers.

La ventaja de esta decisión recae que al tener el mismo mecanismo de entrada y salida, los resultados de la técnica pueden ser tratados utilizando el mismo motor de procesamiento y bajo el mismo modelo de arquitectura, permitiendo un análisis o procesamiento incremental o progreso. Dicho de otro modo, se permite continuar el procesamiento del flujo original en

tiempo real, ya que el algoritmo irá actualizando sus datos de salida a medida que lleguen nuevos datos en el flujo original. Por ejemplo, el usuario podría utilizar el nuevo flujo sobre un servidor de visualización de datos (como lightning-viz) para realizar gráficos de los clusters detectados en tiempo real o también para realizar una evaluación de la calidad de los clusters también en tiempo real.

Implementación en Apache Spark Streaming

Antes de empezar a hablar de los detalles de implementación, la primera elección tomada a la hora de empezar a desarrollar sobre Spark Streaming, fue elegir entre los lenguajes Scala, Java y Python. Aunque Python y Java son dos de los lenguajes más utilizados, se decidió llevar a cabo el desarrollo utilizando el lenguaje Scala debido a los siguientes motivos:

- El framework Spark está escrito en Scala, y por lo tanto, es más fácil interactuar con el código fuente de Spark al tener al menos, la capacidad de leer el código de Scala. A consecuencia, Spark tiene una mejor documentación en Scala en comparación con las API de Java y Python.

- Los métodos de los RDD se asemejan mucho a los de la API de colecciones en Scala. Las funciones de RDD, como map, filter, flatMap, reduce y fold, tienen especificaciones casi idénticas a sus equivalentes de Scala.

- Spark es un framework funcional que se basa en gran medida en conceptos como inmunidad y definiciones lambda, por lo que utilizar la API de Spark puede ser más intuitivo y recomendable sobre un lenguaje con características funcionales como lo es Scala.

- Spark cuenta con una shell que es muy útil para desarrollar y depurar las aplicaciones, y sólo esta disponibles para lenguajes que tengan una consola REPLs como es el caso de Scala y Python.

- Puede ser atractivo desarrollar programas en Spark utilizando Python, ya que es fácil de aprender, rápido de escribir, interpretar e incluso tiene un conjunto muy amplio de herramientas para data science. Sin embargo, el código Spark escrito en Python suele ser más lento que el código equivalente escrito en la JVM, ya qué Scala está tipado estáticamente, y el costo de la comunicación JVM (de Python a Scala) puede ser muy alto. Por último, las funciones de Spark generalmente se escriben primero en Scala y luego se traducen a Python, por lo que para utilizar la funcionalidad Spark

de vanguardia, se deberá trabajar sobre la JVM, es decir, utilizando Scala o Java. Por ejemplo, el soporte de Python para MLlib y Spark Streaming está bastante atrasado en funcionalidades con respecto a la API de Scala.

Luego de elegir Scala como lenguaje de programación, se empezó a modelar el diseño del algoritmo sobre este lenguaje. Scala al ser un lenguaje que combina tanto la programación orientada a objetos como la programación funcional, nos permite modelar los componentes de nuestro algoritmo como objetos y aprovechar las ventajas de ambos paradigmas. Por lo tanto, la implementación del algoritmo está conformado por una clase principal encargada de encapsular toda la lógica de la técnica, la cual recibe el nombre de D3CAS.

Al seguir el enfoque de dos fases, Online-Offline, y ya al tener definido en la sección anterior que tareas tiene que realizar cada una, se tomó la decisión que los componentes o fases se implementen como clases distintas, donde cada uno tiene su comportamiento y estado propio, permitiendo encapsular, modularizar y desacoplar el procesamiento de cada fase, lo que también permitió dejar una distinción clara de qué tareas serán ejecutadas en los nodos workers y cuales en el nodo master. Los nombres de clase asignados son: OnlinePhase y OfflinePhase. Entonces, la clase principal, estará compuesta por objetos de estas clases y específicamente, el objeto de la clase OnlinePhase consumirá los datos proveniente del flujo de datos y los nodos workers serán los encargados de utilizar este objeto. Luego los micro-clusters generados serán consumidos y procesados por el objeto de la clase OnlinePhase en el nodo master.

Como se mencionó en uno de los capítulos anteriores, los flujos de datos en Spark Streaming se representan por medio de la clase DStream, los cuales al seguir las características funcionales de los RDD, son inmutables, por lo que cada operación de transformación que se realice sobre el flujo producirá como resultado un nuevo flujo de datos. Debido a esta característica, en la implementación de la técnica se puede observar que en cada fase o etapa, se procesa una representación distinta del flujo, hablando técnicamente, durante la ejecución se procesan distintos tipos de DStream, esto se puede observar gráficamente a continuación:

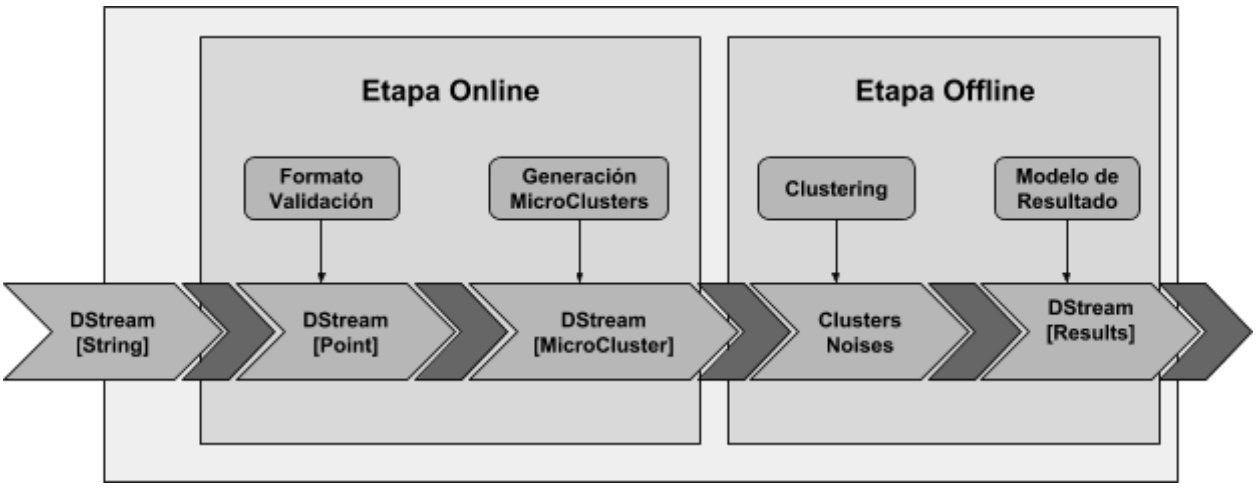


Figura 4.4: Transformaciones del Flujo de dato.

Como se puede observar en el gráfico anterior, antes de cualquier transformación el flujo de entrada original se representa por medio de un flujo de Strings, en Spark Streaming esto está modelado por DStream[String].

Luego en la fase online, en la primera tarea de validación y formato, se lleva a cabo el análisis de los strings de entrada transformándolos a la estructura básica de procesamiento que son los objetos Puntos, es por esto, que al final de esta tarea se genera como salida un flujo de puntos, representado en Spark como DStream[Point], que rápidamente serán consumidos por la tarea de generación de microClusters, produciendo como salida un DStream[Micro-Cluster] representando los micro-clusters para los datos actuales en el flujo.

A continuación, en la fase offline, el nodo master será el encargado de consumir la información del flujo de micro-clusters con el objetivo de llevar a cabo el proceso de actualización y ejecutar la tarea de clustering. Una vez terminada esta tarea, se tienen tanto los modelos de clusters actuales como también la representación de los elementos que son outliers o ruido. Por último, debido a la política elegida para la representación de resultados descrita anteriormente, la salida de la fase offline y a su vez la del algoritmo propuesto debe estar representada por un flujo de datos que representan los resultados para cada intervalo de tiempo procesado, por lo tanto, se genera como salida un DStream[Resultados], el cual es un flujo donde cada elemento es una estructura o objeto que contiene la información de las agrupaciones producidas (identificación + centro del cluster junto a los puntos virtuales que conforman cada cluster) y la representación de los elementos outliers.

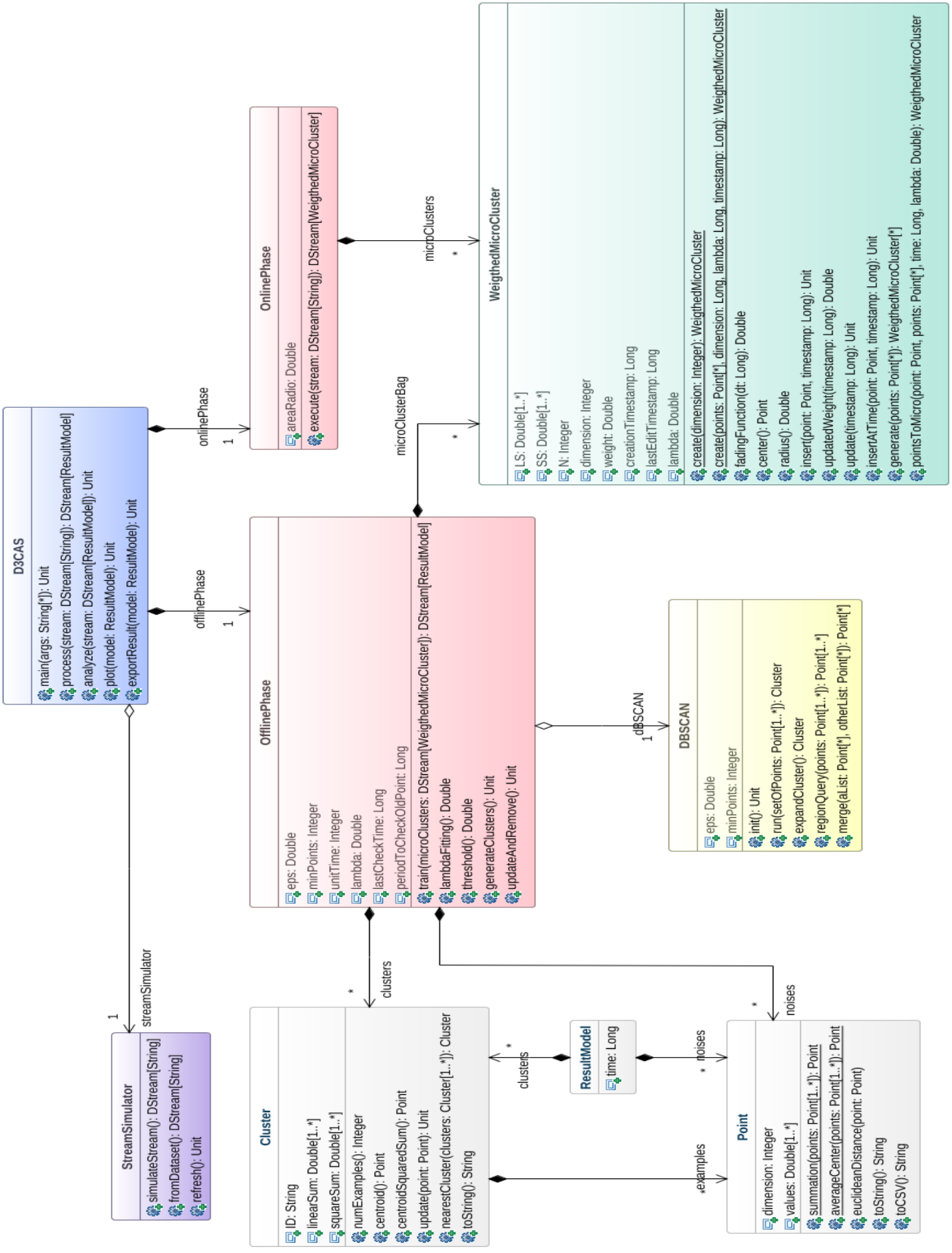
Para finalizar este capítulo se muestra en la tabla 4.1 un cuadro comparativo de las ventajas y desventajas de D3CAS versus los algoritmos estudiados en esta tesina, el diagrama de clases UML de la implementación de D3CAS en la figura 4.5, y por último, los pseudocódigo del proceso driver (figura 4.6), de la fase online (figura 4.7) y de la fase offline (Figura 4.8).

Cuadro Comparativo

	BIRCH	Clustream	Clustree	DenStream	D3CAS
Enfoque	online-offline	online-offline	online-offline	online-offline	online-offline
Estructura	feature vectors	coreset	Tree-R (feature vectors)	feature vectors	feature vectors
Ventana	Landmark	Landmark	Damped	Damped	Damped
Algoritmo clustering	K-means	K-means	K-means	DBSCAN	DBSCAN
Detección formas esféricas	✔	✔	✔	✔	✔
Detección formas arbitrarias			✔	✔	✔
Detección ruido				✔	✔
Concept Drift			✔	✔	✔
Distribuido					✔

Tabla 4.1: Tabla comparativa entre D3CAS y los algoritmos presentados en el capítulo 3.

Figura 4.5: Diagrama UML D3CAS



Pseudo código D3CAS

Algoritmo 1: D3CAS

```
Input: stream: DStream[String] /* stream puede ser infinito  */
Output: streamResults: DStream[ResultModel]
1 let offlinePhase ← new OfflinePhase();
2 let onlinePhase ← new OnlinePhase();
3 let streamResults ← streamingContext.get();
4 foreach (batch in stream) do
5 |   let microClusters ← onlinePhase.execute(batch);
6 |   let results ← offlinePhase.train(microClusters);
  |   /* emitir los nuevos resultados en el stream de salida */
7 |   streamResults.stream(results);
8 end
```

Figura 4.6: Pseudocódigo D3CAS.

Procedimiento onlinePhase.execute(batch)

```
Input: batch: DStream[String] /* batch contiene los puntos que
         llegan al stream en un intervalo de tiempo.          */
Output: microClustersList: DStream[MicroCluster]
1 let radio_e ← number;
2 let pointList ← new List();
3 let microClusterList ← new List();
4 foreach (data in batch) do
5 |   pointList ← pointList + transformStringToPoints(data);
6 end
  /* Generar los micro-clusters del batch actual            */
7 while (pointList.length ≠ 0) do
8 |   let first ← pointsList.first();
9 |   let neighbors ← first.getPointNeighbors(radio_e, pointList);
10|   let microCluster ← generateMicroCluster(first, neighbors);
11|   microClusterList ← microClusterList + microCluster;
12|   pointsList ← pointsList - first;
13|   pointsList ← pointsList.removeAll(neighbors);
14 end
15 return streamingContext.stream(microClustersList);
```

Figura 4.7: Pseudocódigo fase online.

Procedimiento offlinePhase.train(stream)

Input: microClusters: DStream[MicroCluster] `/* contiene listas de micro-clusters que generan los workers en la etapa online */`
Output: results: ResultModel

```
1  let eps ← number;
2  let minPoints ← number;
3  let microClustersBag ← new List();
4  let time ← 0;
5  let lambda ← number;
6  let clusters ← new List();
7  let noise ← new List();
8  foreach (e in microClusters) do
9  |  time ← e.getTimeContext();
10 |  let micros ← e.getMicroClusterList();
11 |  microClustersBag ← microClustersBag.addAll(micros);
12 end
13 if (fadingWindow.timeToUpdate(time)) then
14 |  foreach (mc in microClustersBag) do
15 |  |  mc.updateWeigth(time, lambda);
16 |  |  if (mc.wiegth < fadingWindow.threshold) then
17 |  |  |  microClustersBag ← microClustersBag - mc;
18 |  |  end
19 |  end
20 end
21 let centers ← microClustersBag.toPoints();
22 (clusters, noise) ← DBSCAN.run(centers, eps, minPoints);
23 return ResultModel(clusters, noise);
```

Figura 4.8: Pseudocódigo fase offline.

Capítulo 5

Evaluación y comparación

Conceptos para la validez de agrupaciones

El procedimiento de evaluación de la calidad de los resultados de un algoritmo de agrupamiento se conoce bajo el término **validez de clúster**[47]. En términos generales, la evaluación de validez de clúster tratan de demostrar que tan diferentes son los grupos formados. Para llevar a cabo estos procedimientos, existen tres enfoques para estudiar la validez de los resultados de clustering [48].

El primero se basa en **criterios externos**. Esto implica que los resultados de un algoritmo de agrupamiento se evalúan basados sobre una estructura pre-especificada, que se indica junto al conjunto de datos y refleja la intuición sobre el modelo de agrupamiento del conjunto de datos, es decir, se comparan los resultados contra una estructura/modelo de agrupación conocida del conjunto (por ejemplo, por medio de etiquetas, *labels*, de identificación de grupo para cada elemento del conjunto de datos).

El segundo enfoque se basa en **criterios internos**. Los criterios internos se utilizan para medir la calidad de una estructura de agrupamiento sin utilizar información externa, por lo tanto, la evaluación se lleva a cabo en función de los datos agrupados, es decir, por medio de las características inherentes presentes en el modelo generado. Los índices utilizados para criterios internos basan su evaluación por medio de la similitud entre los elementos de un grupo y la disimilitud entre grupos, por lo que estos índices asignan un valor bueno a los algoritmos que producen agrupaciones con gran similitud dentro de cada clúster y baja similitud entre los distintos clusters. Por lo tanto, las medidas de evaluación interna son las más adecuadas para obtener en un determinado contexto una comparación en las que un algoritmo funciona mejor que otro, pero esto no implica que un algoritmo produzca resultados más válidos que otro.[50]

El tercer enfoque de validez se basa en **criterios relativos**. Aquí la idea básica es la evaluación de una estructura de agrupación comparándola con otros modelos de agrupación, resultantes por el mismo u otro algoritmo con o diferentes valores para los parámetros.

A la hora de llevar a cabo el análisis de los clusters, para la evaluación de clusters y para la selección de un modelo de agrupamiento óptimo se analizan dos características importantes [49]:

- Compactness (compactibilidad, compactación o compacidad): la cual indica la cercanía entre todos los miembros que forman un grupo. Para un buen valor de *compactness* se debe cumplir que los miembros de cada grupo deben estar lo más cerca posible entre ellos. Una medida común de *compactness* es la varianza.

- Separation (Separación): la cual indica la separación entre los grupos formados. Para un buen valor de separación se dará cuando los grupos formados estén ampliamente separados. Hay tres enfoques comunes que miden la distancia entre dos grupos diferentes:

    - Single linkage (Enlace único): mide la distancia entre los miembros de los clusters más cercanos.
    - Complete linkage (Enlace completo): mide la distancia entre los miembros de los clusters más distantes.
    - Comparison of centroids (Comparación de centroides): mide la distancia entre los centros de los clusters.

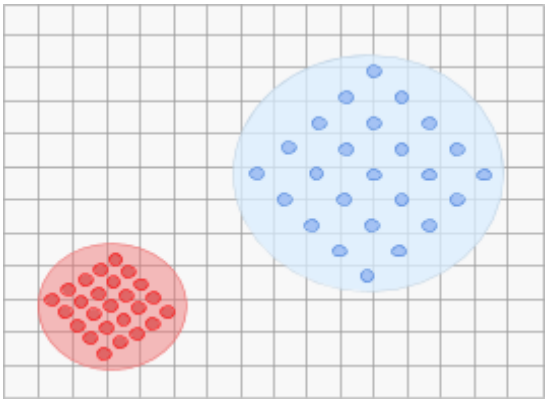


Figura 5.1: Ejemplo de compactación: El cluster rojo es un cluster más compacto que el cluster Azul.

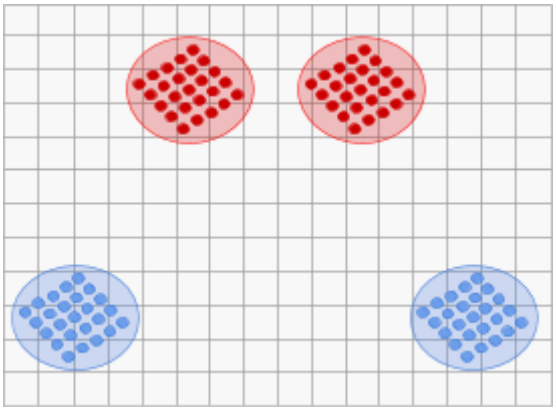


Figura 5.2: Ejemplo de separación: Los clusters azules están más separados que los cluster rojos.

Silhouette

Silhouette es un enfoque de validación de criterios internos y es el que será utilizado en las evaluaciones y comparaciones de esta tesina. Al utilizar Silhouette cada grupo está representado por una estructura *'silueta'*, que se basa en la comparación de compacidad y separación de cada grupo. Cada silueta muestra que tan bien agrupado está cada objeto dentro de su grupo, es decir, cuán similar es un objeto a su propio clúster en comparación con otros clústeres.

El valor de silueta es una medida/índice de calidad numérica que va entre -1 y +1, donde una buena calidad está representado por un valor alto (más próximo a +1) indicando que el objeto está bien agrupado en su cluster y está poco relacionado con los clústeres vecinos. Mientras que los valores más próximos a -1 indican una baja calidad debido a que el objeto no debería pertenecer al grupo, ya sea porque el elemento pertenece a otro grupo, es un outlier o porque el grupo actual está mal formado en cuanto a las características de compacidad y separación.

El agrupamiento global de un dataset se puede visualizar en un único diagrama combinando los valores s(i) de cada elemento i de todos los clusters, lo que permite una apreciación de la calidad relativa de los clústeres y una visión general de la configuración de los datos (figura 5.3). El ancho promedio de la silueta proporciona una evaluación de la validez de la agrupación y se puede usar para seleccionar un número "apropiado" de clústeres. Generalmente, el ancho promedio de las siluetas se encuentra representado por el centroide de cada grupo.

Definición de Silhouette

Silhouette se puede calcular con cualquier medida de distancia, como la distancia euclidiana o la distancia de Manhattan y es recomendable utilizarlo cuando se buscan grupos compactos y claramente separados.

Luego de llevar a cabo la agrupación y generar el modelo de agrupación el cual está formado por los datos distribuidos en distintos grupos, el método de Silhouette se lleva a cabo de la siguiente forma, para cada dato i del dataset:

- $a(i)$. Sea $a(i)$ la distancia promedio entre el elemento i y todos los elementos que pertenecen al mismo cluster A, a la distancia promedio se la denomina '*disimilitud promedio*' y cuanto menor sea la distancia promedio, mejor es la asignación. Por lo que se puede interpretar a $a(i)$ como una medida de qué tan bien asignado está el elemento i a su cluster.

- Luego, para cada cluster C distintos de A, se calcula la '*disimilitud promedio*' del elemento i al cluster C, la cual está dada por el promedio de las distancias entre el elemento i y todos los puntos pertenecientes a C. Después de calcular la disimilitud promedio para cada cluster C, se define $b(i)$ como la disimilitud promedio más baja calculada con respecto al elemento i. Se dice que el clúster con la disimilitud promedio más baja es el "clúster vecino" de i o el cluster competidor de A porque es el siguiente clúster que mejor se ajusta a los valores de i.

Una vez encontrado $a(i)$ y $b(i)$, el índice de silhouette para el elemento *i* se calcula como:

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

también se puede definir como:

$$s(i) = \begin{cases} 1 - a(i)/b(i), & \text{if } a(i) < b(i) \\ 0, & \text{if } a(i) = b(i) \\ b(i)/a(i) - 1, & \text{if } a(i) > b(i) \end{cases}$$

Dada esta definición se puede observar que valor de silhouette se encuentra entre -1 y +1

$$-1 \leq s(i) \leq 1$$

Cuando el cluster A contiene solamente un objeto, no es posible aplicar el método para calcular $a(i)$, por lo que simplemente se asigna $s(i) = 0$.

Cuando $s(i)$ está cerca de +1 entonces se puede afirmar que $a(i) < b(i)$. Como $a(i)$ es una medida que indica que tan diferente es el objeto con respecto a los de su cluster asignado, un valor pequeño significa que el elemento está bien ubicado en su cluster. Además, un valor grande de $b(i)$ implica que el elemento i no estaria bien asignado en su cluster vecino. Por lo tanto, un valor de $s(i)$ cercano a +1 significa que el elemento i está agrupado apropiadamente en el cluster A.

Ahora cuando $s(i)$ está cerca de -1, se da que $a(i)$ es mucho más grande que $b(i)$, y siguiendo la lógica anterior, se puede afirmar que el elemento i sería más apropiado que esté agrupado en el su cluster vecino, ya que el elemento i se encuentra mucho más cerca del cluster B que del A. Por lo tanto, podemos concluir que en este caso el elemento fue mal agrupado.

Por último, cuando el valor de $s(i)$ está cerca de 0, significa que el elemento i esta en el borde de dos clusters que podrían agrupar a i, es decir, no esta claro en que cluster estaría correctamente asignado. A continuación se da un ejemplo de cómo es la visualización de este método.

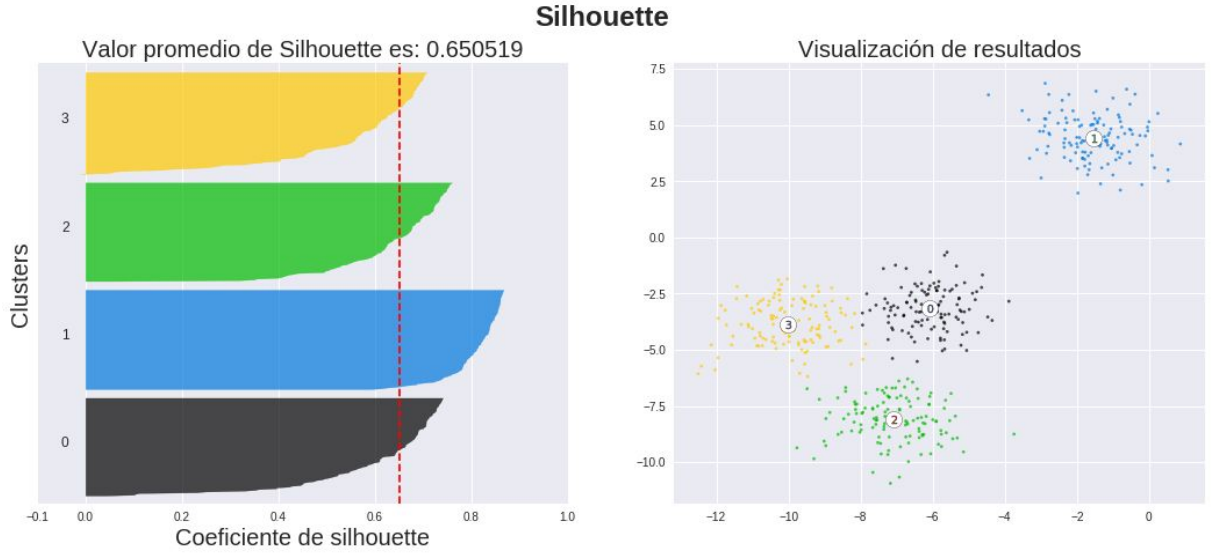


Figura 5.3: Ejemplo de Silhouettes aplicado a todos los elementos

Como conclusión final, el valor promedio de $s(i)$ de todos los datos dentro de un cluster indica la medida de que tan bien agrupado están todos los datos dentro de ese cluster. Por lo tanto, un valor promedio de $s(i)$ de los $s(i)$ de todos los clusters detectados es una medida de que tan bueno fue el agrupamiento realizado. A partir de esto, podemos destacar

que una representación compacta de coeficiente de Silhouette es utilizando un gráfico de barra donde se muestran los índices promedio para cada grupo. Por ejemplo, para el caso anterior se lo puede representar como se grafica en la figura 5.4:

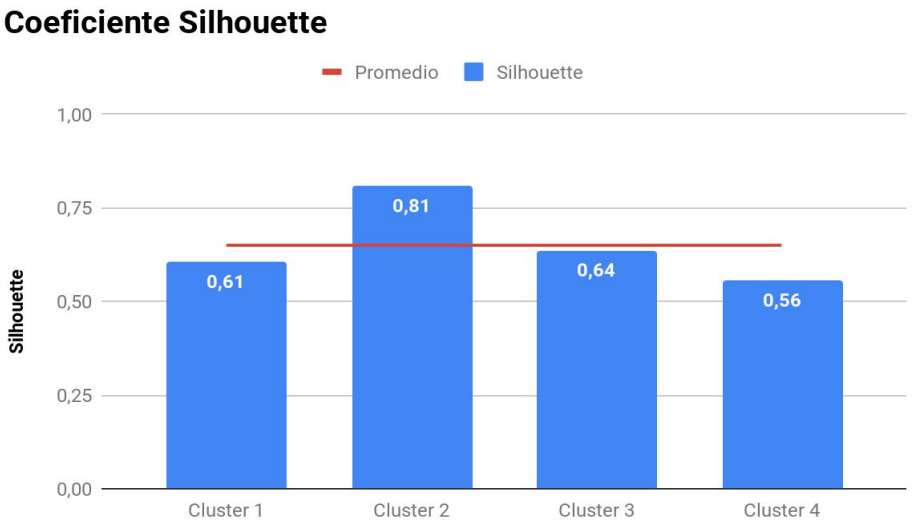


Figura 5.4: Ejemplo de Silhouettes promedio por cluster

Evaluaciones y comparaciones

Los experimentos, pruebas y evaluaciones realizadas en este trabajo, fueron ejecutadas sobre un ambiente local, es decir, utilizando una única computadora. Dicha computadora cuenta con un procesador Intel i5 3210M, RAM de 8GB, una unidad de estado sólido (SSD) de 240 gb y utilizando un sistema operativo Linux, específicamente Ubuntu 16.10. En cuanto a la configuración de Spark se utilizó la versión 2.2.0 tanto para el core API de Spark como para la extensión de Spark Streaming, las cuales necesitan tener configurado la versión 2.11.8 de Scala.

Como las pruebas realizadas fueron llevadas a cabo en un entorno local, el objetivo principal de estas es medir la calidad de los resultados obtenidos más que hacer un análisis de rendimiento y eficiencia del proceso de ejecución.

Detección dinámica

Como el objetivo específico del algoritmo de D3CAS es la detección de cluster de forma dinámica, se decide en esta etapa como primer medida evaluar esta característica, para esto se evalúa la calidad de los clusters detectados sobre un flujo de datos donde en distintos instantes en el tiempo, el flujo posee un número distintos de agrupaciones.

Para esta prueba al flujo se lo denominara como 100K10C, ya que utiliza un datasets generados por el script GENERATEDATA[55][56], que contiene 100.000 elementos que se encuentran agrupados en 10 clusters, donde cada elemento está constituido por dos atributos continuos. 100K10C se encuentra ordenado por cluster, es decir, que los elementos pertenecientes a un cluster se encuentran juntos/contiguos. Esto último, en el contexto de un flujo de dato, significa que a medida que se procesan los datos perteneciente a un grupo de clusters, al flujo llegan nuevos datos que no pertenecen a los clusters detectados sino que pertenecen a clusters nuevos, lo cual produce que la distribución de los datos vaya cambiando a lo largo del tiempo.

En cuanto a la velocidad de flujo, es decir, la cantidad de elementos que se transmiten en un determinado periodo de tiempo, se estableció que se emitirán 10.000 elementos cada 5 segundos, y entre cada periodo se calcula el coeficiente de silhouette de cada cluster detectado para evaluar la calidad de los grupos formados.

En cuanto a los parámetros del algoritmo D3CAS se utilizaron los siguientes: Para el componente online (capítulo 4, sección “Generación micro-cluster”), se estableció el parámetro de radio $e = 2.0$.

En cuanto al componente offline (capítulo 4, sección “Agrupación basa en densidad”), se establecieron los parámetros $eps = 3.0$ y $minPoints = 4.0$, los cuales se corresponden con los parámetros originales de DBSCAN (eps = ε y minPoints = μ)[35], donde eps representa el radio de un punto i para buscar sus puntos vecinos y $minPoints$ representa la cantidad mínima de puntos vecinos en el radio eps que tiene que tener el punto i para considerarse un cluster. Y por último para controlar la ventana deslizante o fadding windows se utilizaron los parámetros $\lambda = 0.25$ y $\mu = 10$ qué son los parámetros que se utilizan generalmente en este tipo de ventanas [14] (capítulo 4, sección “Actualización temporal”). Estos parámetros de ventana se repiten para todas las pruebas hechas en este capítulo.

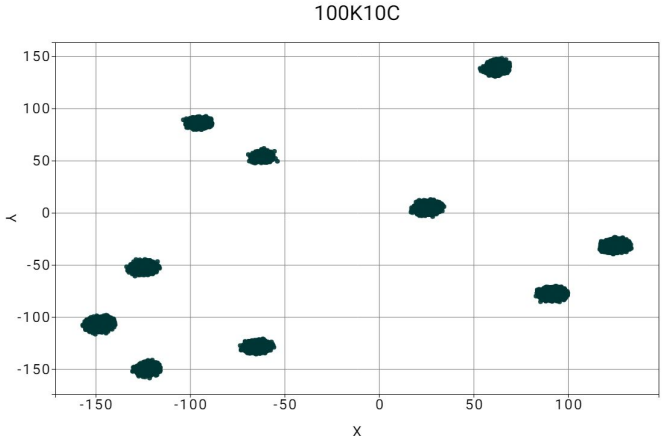


Figura 5.5: ilustración dataset 100K10C

Los resultados obtenidos para este experimento con D3CAS son:

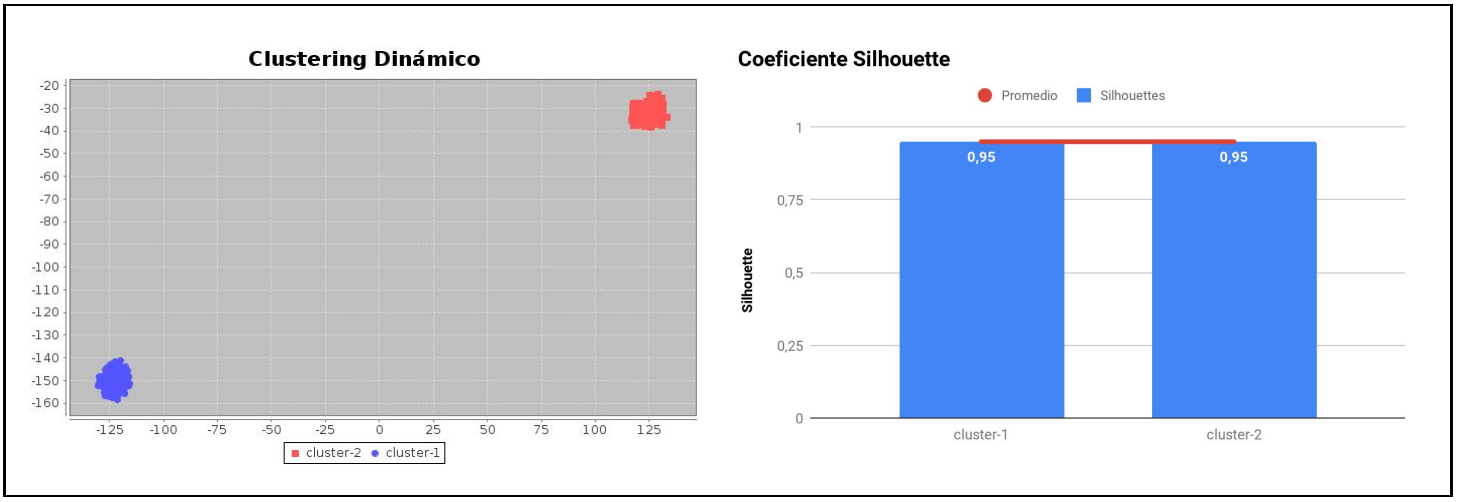


Figura 5.6: Silhouettes en el 1° y 2° periodo de 5 segundo del flujo. Silhouettes promedio = 0.95

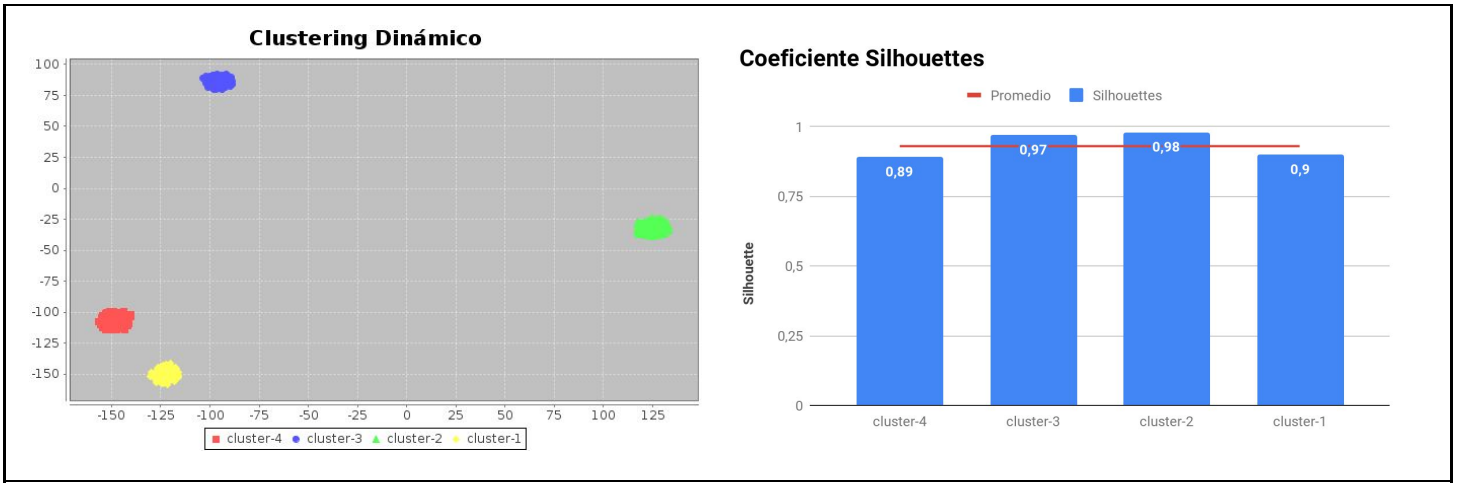


Figura 5.7: Silhouettes en el 3° y 4° periodo de 5 segundo del flujo. Silhouettes promedio = 0.93

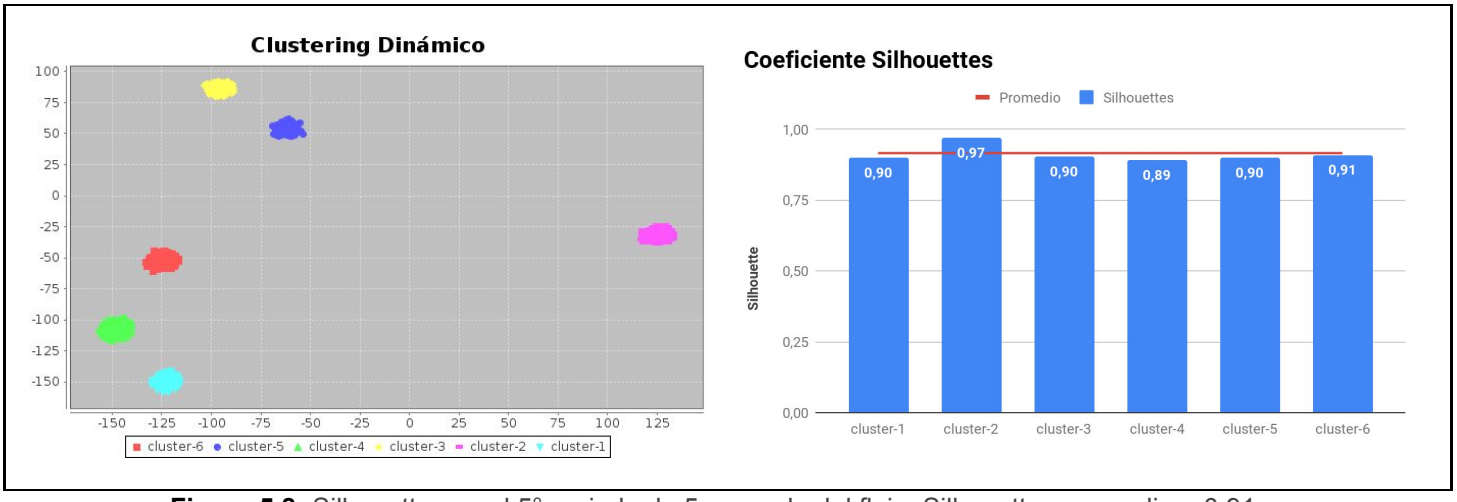


Figura 5.8: Silhouettes en el 5° periodo de 5 segundo del flujo. Silhouettes promedio = 0.91

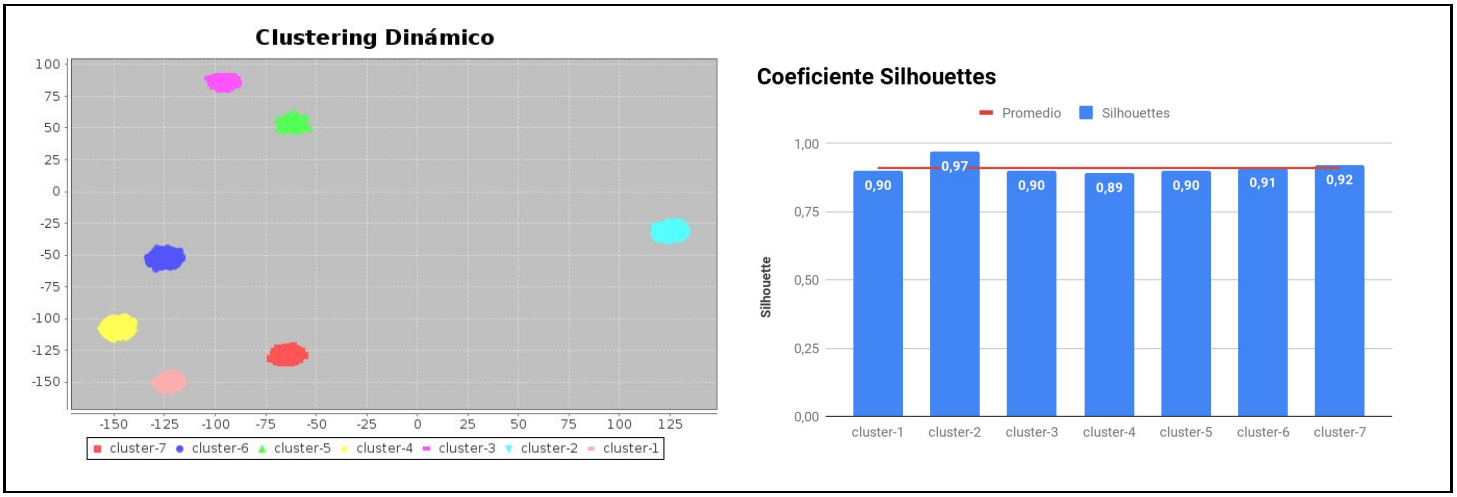


Figura 5.9: Silhouettes en el 6° periodo de 5 segundo del flujo. Silhouettes promedio = 0.91

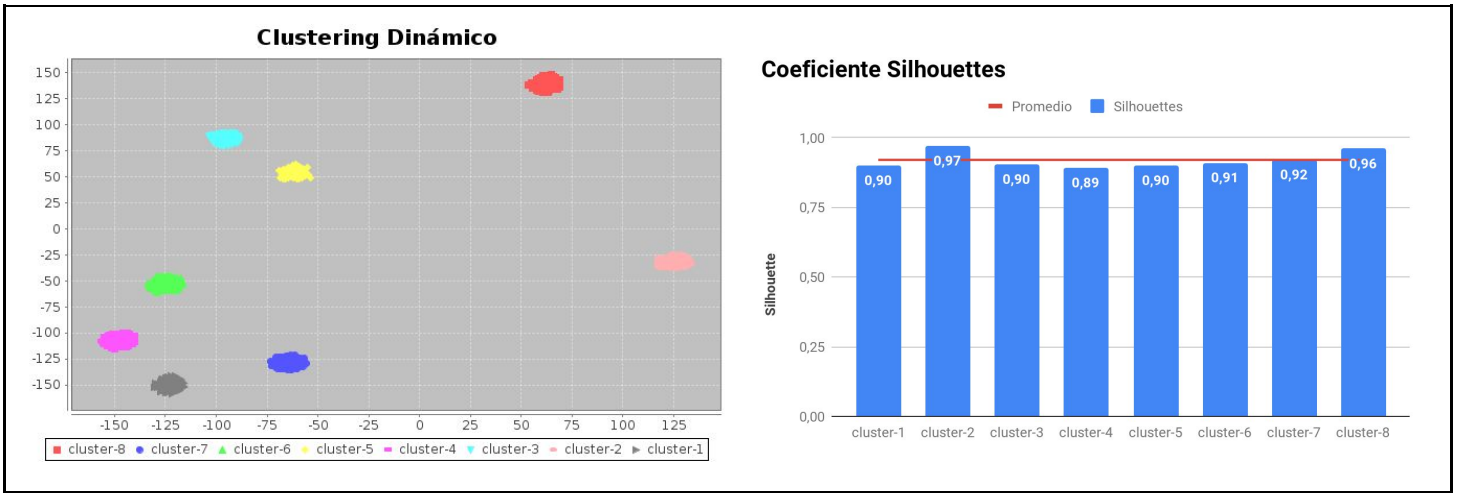


Figura 5.10: Silhouettes en el 7° periodo de 5 segundo del flujo. Silhouettes promedio = 0.92

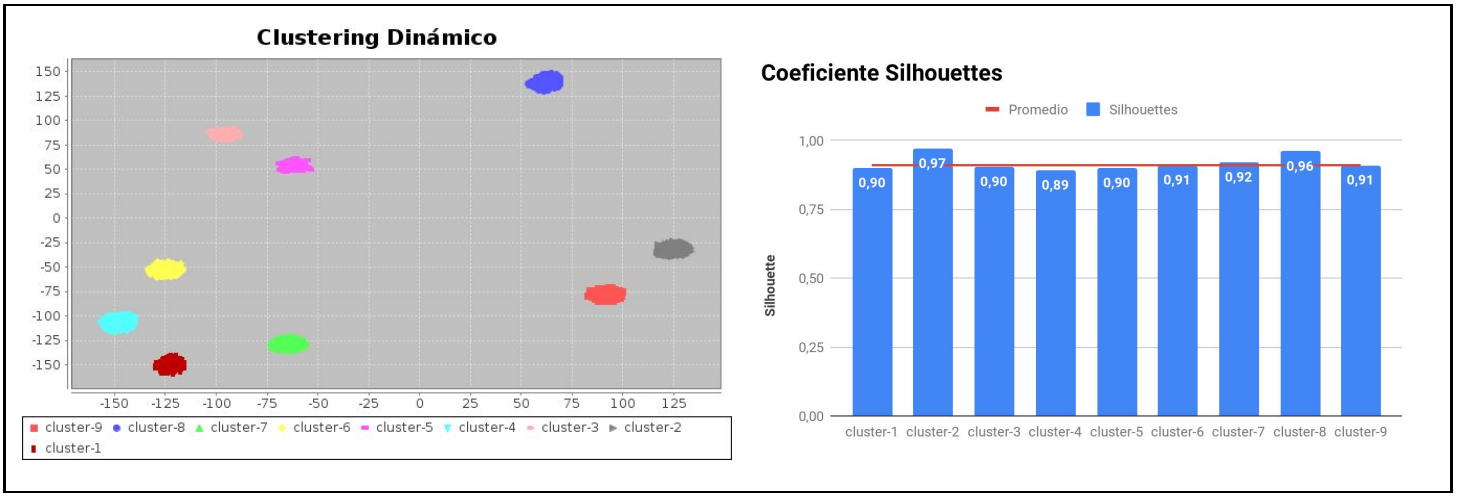


Figura 5.11: Silhouettes en el 8° periodo de 5 segundo del flujo. Silhouettes promedio = 0.91

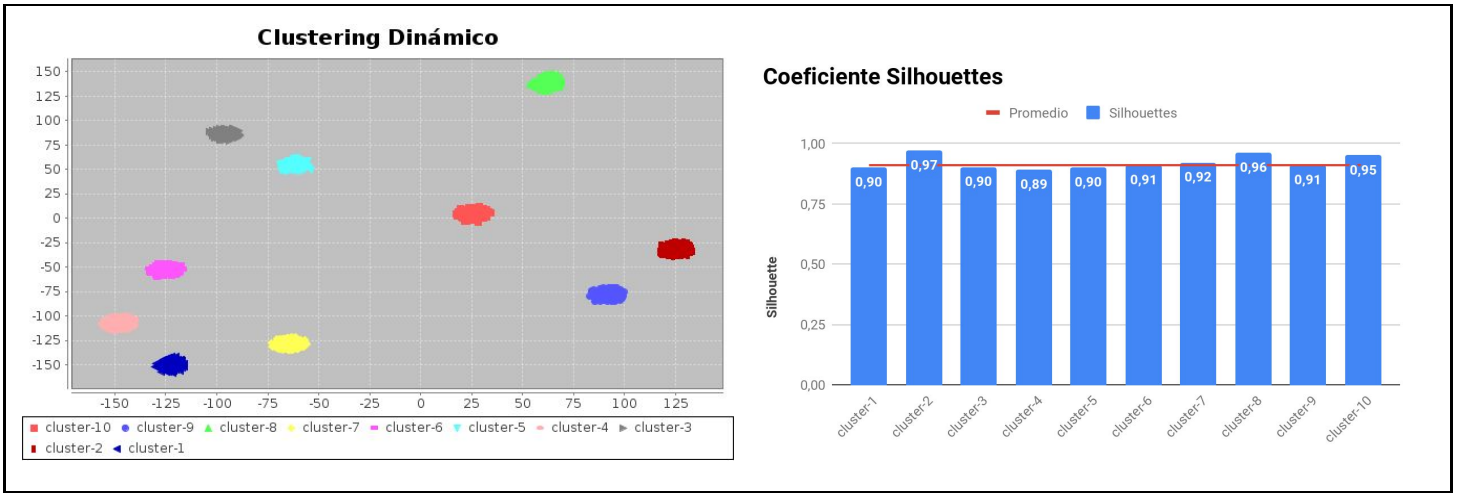


Figura 5.12: Silhouettes en el 9° y 10° periodo de 5 segundo del flujo. Silhouettes promedio = 0.91

Como se puede ver en todos los casos, el índice promedio de Silhouette no baja de 0.91, lo cual indica que al estar cerca del valor +1, cada elemento se encuentra agrupado junto a los elementos que poseen características similares, por lo tanto, el algoritmo está detectando correctamente los distintos grupos que posee el flujo. También se puede destacar que tras la incorporación de nuevos datos en cada periodo, la calidad de las agrupaciones previas no se ven afectadas, ya que no hay una disminución del índice promedio de Silhouette a medida que pasó el tiempo. Tras estos resultados, se puede concluir que el algoritmo es capaz de detectar clusters de forma dinámica safistactoriamente.

Comparación de resultados

Ahora se realizan algunos experimentos de comparación de resultados entre el algoritmo CluStream y D3CAS. Como D3CAS está implementado en Apache Spark, se busco una implementación de CluStream que también estuviese implementado sobre este framework, logrando de esta forma que las comparaciones a realizar se hagan sobre el mismo lenguaje, framework y configuración del entorno de ejecución. La implementación de CluStream utilizada para realizar las comparaciones es una implementación sobre Apache Spark creado por “HUAWEI Noah's Ark Lab” [51] que se encuentra dentro de su proyecto de minería de flujos de datos StreamDM [52].

En primer lugar se realizan pruebas utilizando datasets con distintas cantidades de grupos y que contengan grupos compactos y claramente separados, debido a que son los recomendables para realizar evaluaciones de tipo internas mediante el método del coeficiente de Silhouette. Los datasets a utilizar se generaron utilizando el script GENERATEDATA[55][56]. Para nombrar y diferenciar los datasets generados se definió la siguiente nomenclatura: 10K#C, que representa que el dataset contiene 10.000 elementos y que se encuentran agrupados en un número ‘#’ de cluster, por ejemplo, el dataset 10K4C,

indica que posee 10.000 elementos agrupados en 4 grupos. Todos estos datasets se encuentran ordenados por cluster, es decir, que los elementos pertenecientes a un cluster se encuentran juntos/contiguos.

En cuanto al flujo de datos, se estableció que se emitirán de 1.000 elementos del datasets cada 5 segundos. Por último, se medirá el coeficiente de Silhouette promedio una vez transmitido todos los elementos del dataset. A continuación se listan las pruebas hechas y se muestran en gráficos los resultados obtenidos:

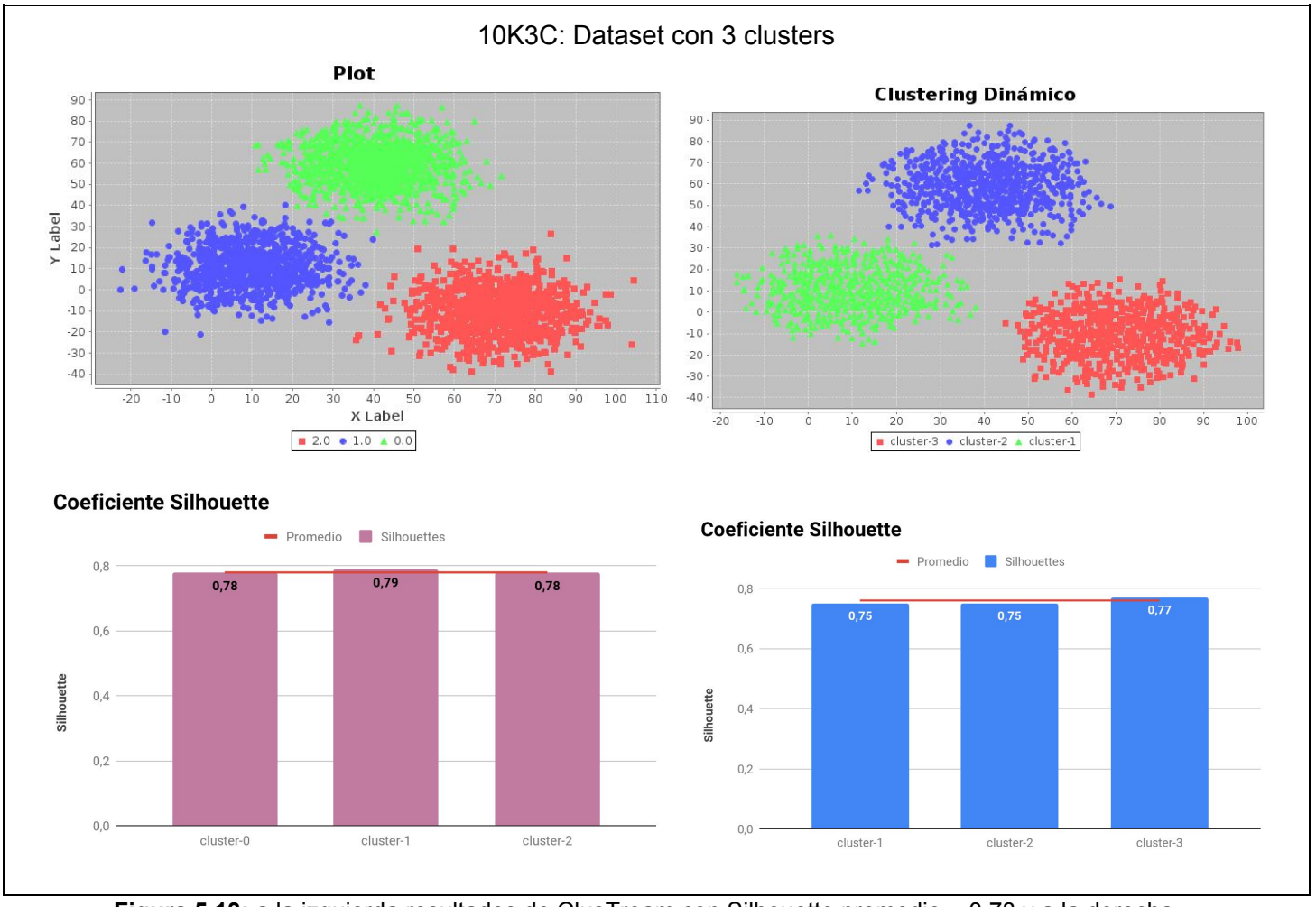


Figura 5.13: a la izquierda resultados de ClusTream con Silhouette promedio = 0.78 y a la derecha resultados de D3CAS con Silhouette promedio = 0.76

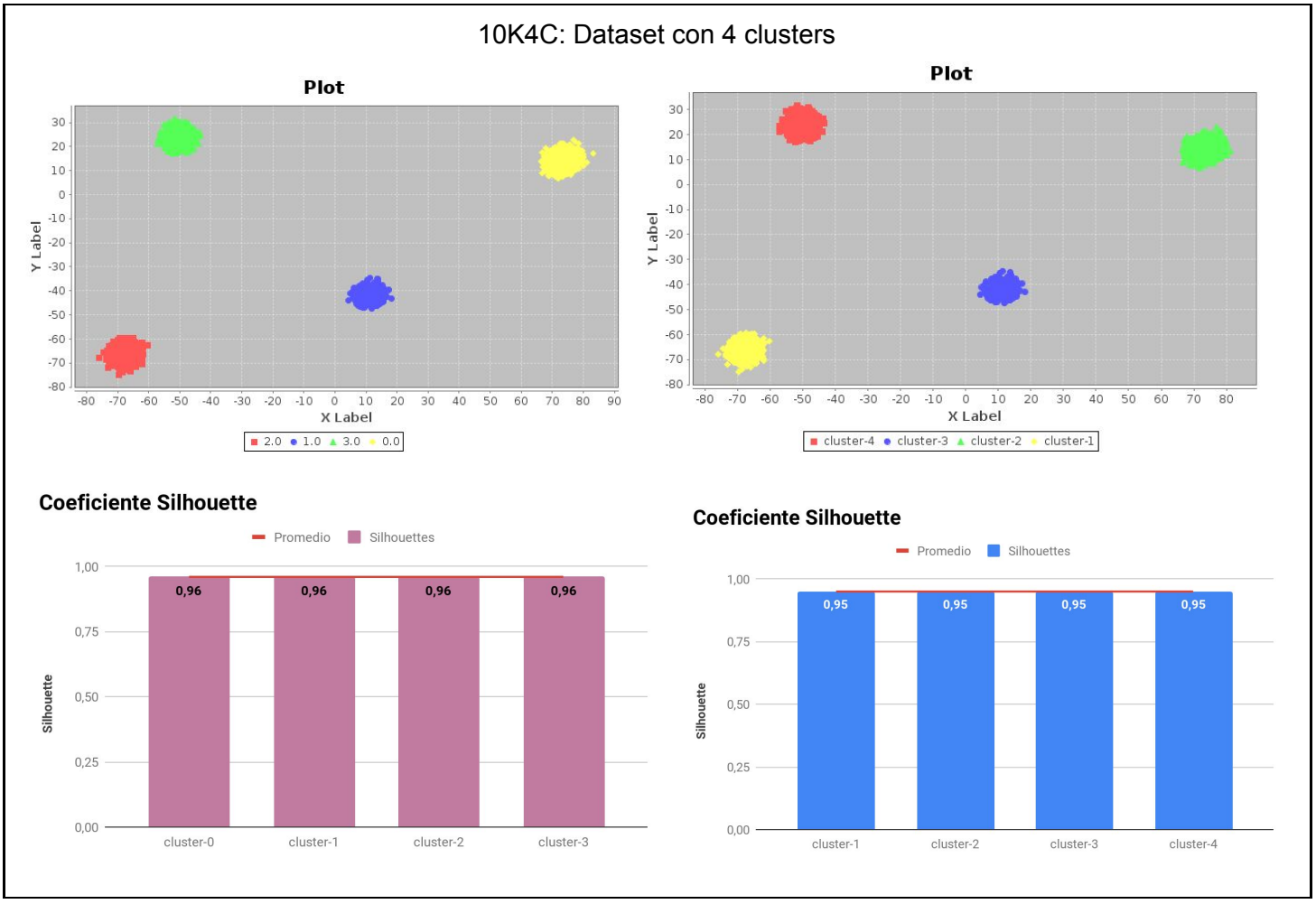
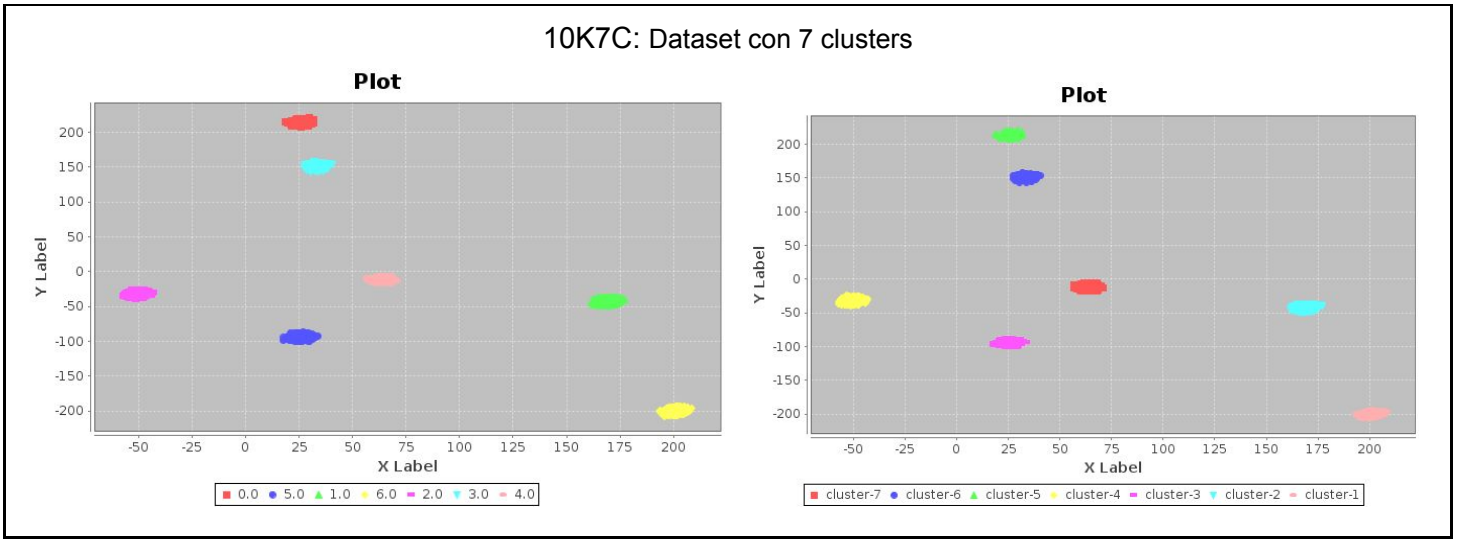


Figura 5.14: a la izquierda resultados de Clustream con Silhouette promedio = 0.96 y a la derecha resultados de D3CAS con Silhouette promedio = 0.95



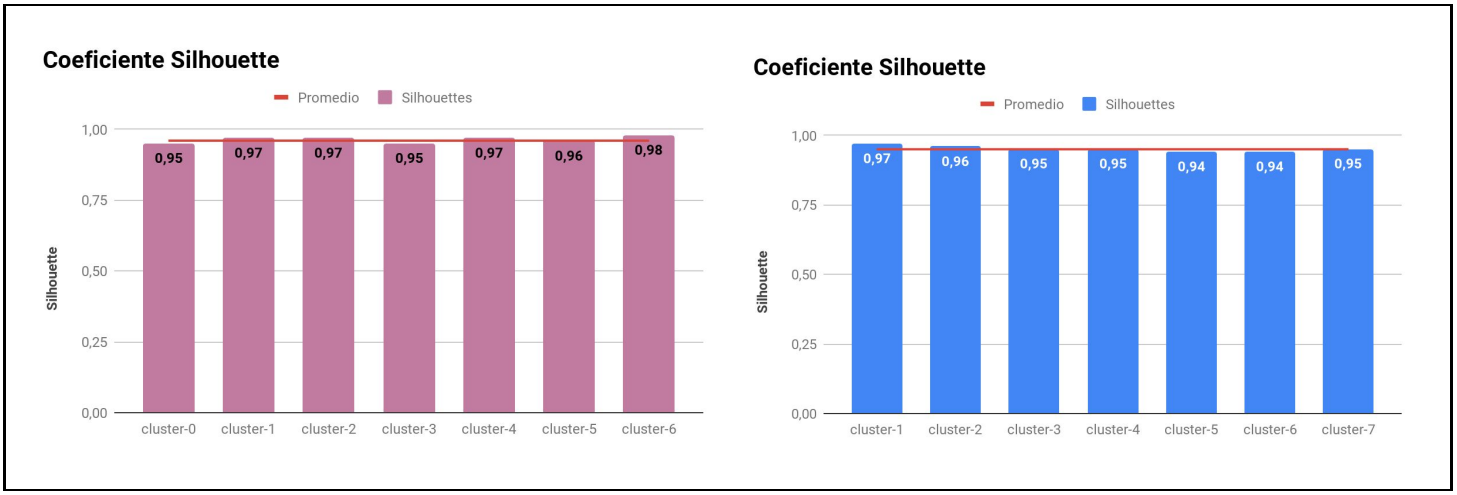


Figura 5.15: a la izquierda resultados de Clustream con Silhouette promedio = 0.96 y a la derecha resultados de D3CAS con Silhouette promedio = 0.95

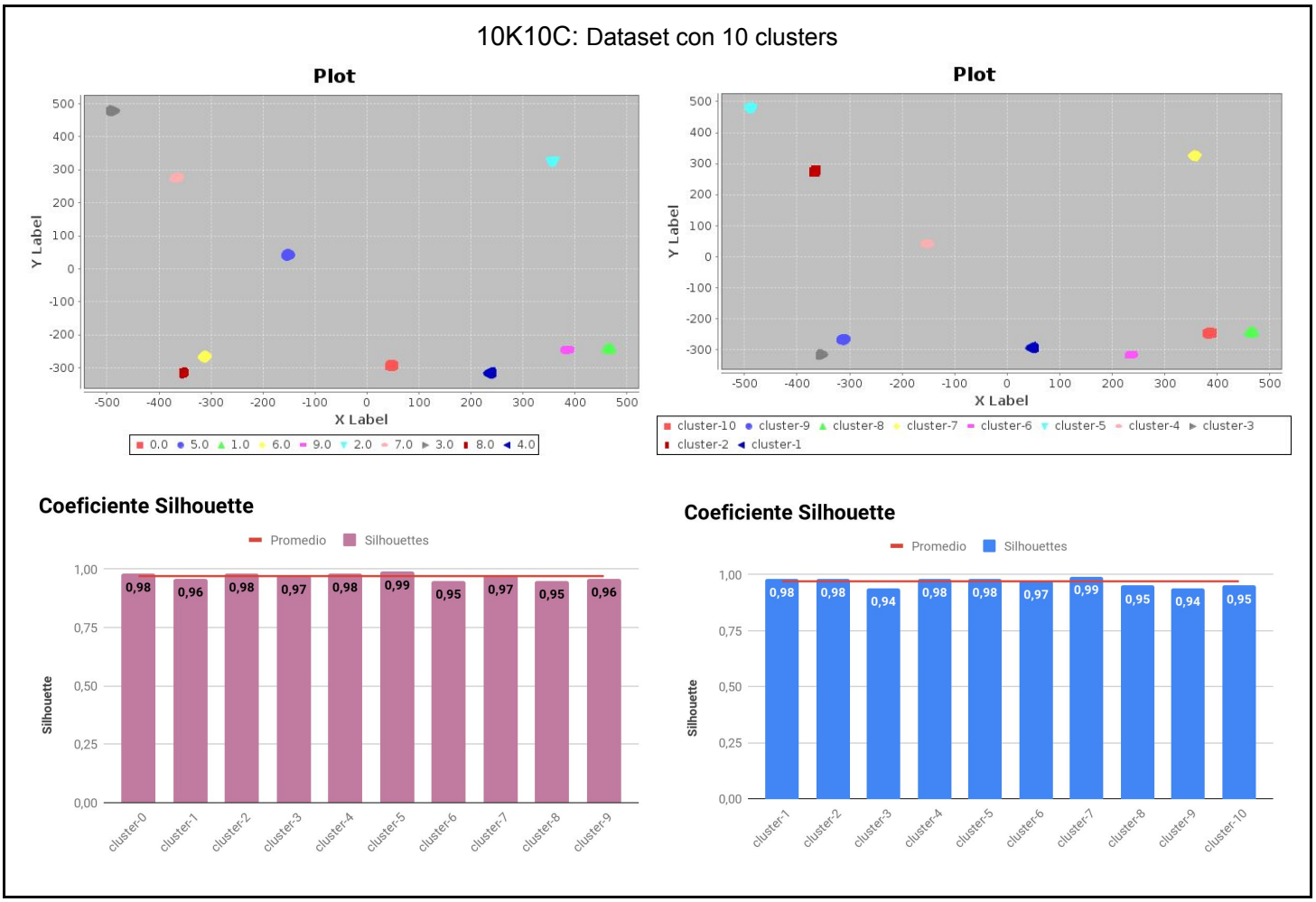


Figura 5.16: a la izquierda resultados de Clustream con Silhouette promedio = 0.97 y a la derecha resultados de D3CAS con Silhouette promedio = 0.97

Como se puede ver en estas comparaciones ambos algoritmos dan muy buenos resultados, sólo en el dataset de 3 clusters (Figura 5.13) dieron valores de Silhouette promedios de 0.78 y 0.76, y en los demás casos dieron un valor por arriba de 0.95, lo cual indica que los clusters detectados están bien formados y separados entre sí. Y si comparamos los resultados de ambos, solamente hay una diferencia de Silhouette entre 0.01 y 0.02, la cual es mínima e imperceptible, sin embargo, dicha diferencia se inclina a favor del algoritmo de CluStream. Pero al elevar el número de clusters a detectar encontramos los siguientes resultados:

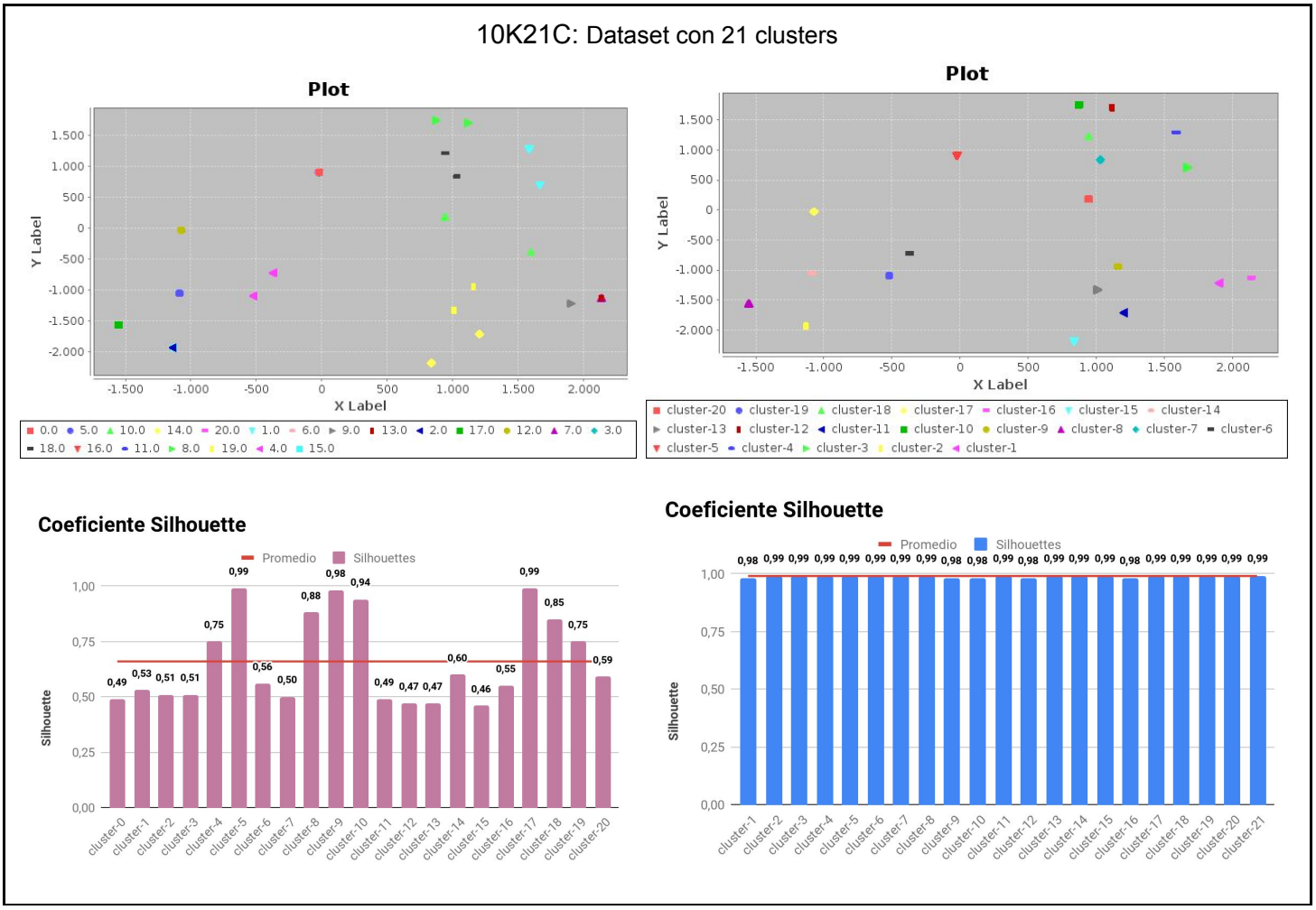


Figura 5.17: a la izquierda resultados de Clustream con Silhouette promedio = 0.66 y a la derecha resultados de D3CAS con Silhouette promedio = 0.99

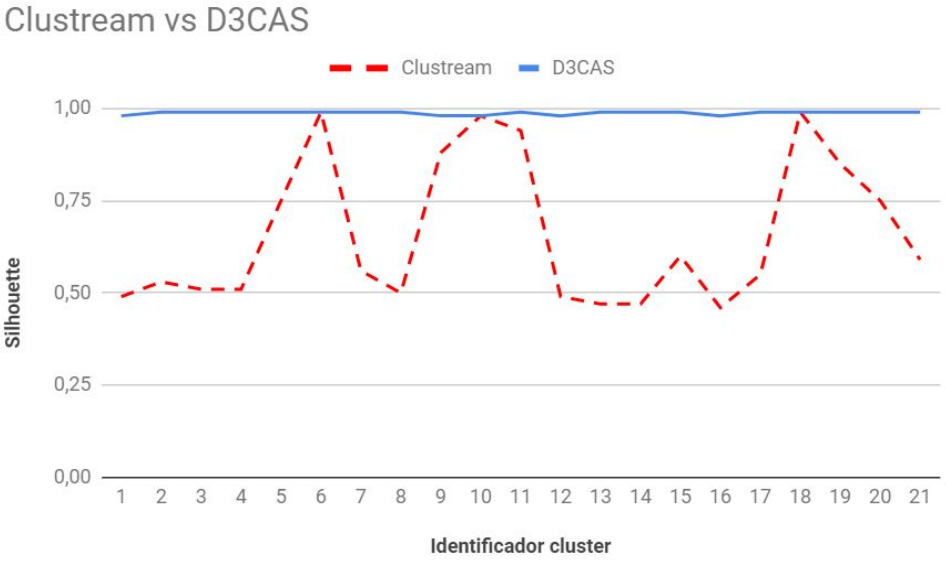


Figura 5.18: Comparación de Silhouette entre Clustream y D3CAS utilizando el dataset de 21 clusters.

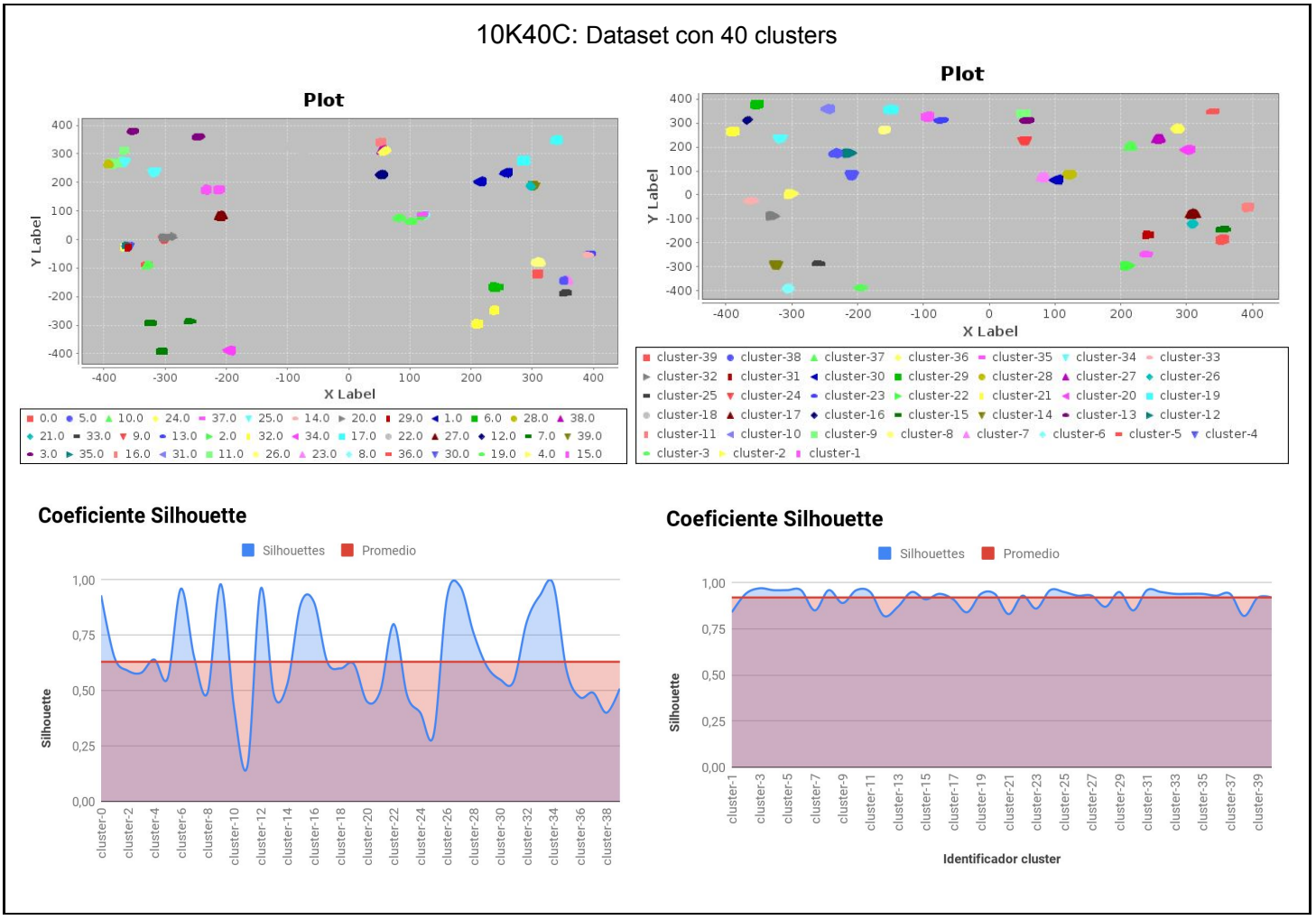


Figura 5.19: a la izquierda resultados de Clustream con Silhouette promedio = 0.63 y a la derecha resultados de D3CAS con Silhouette promedio = 0.92

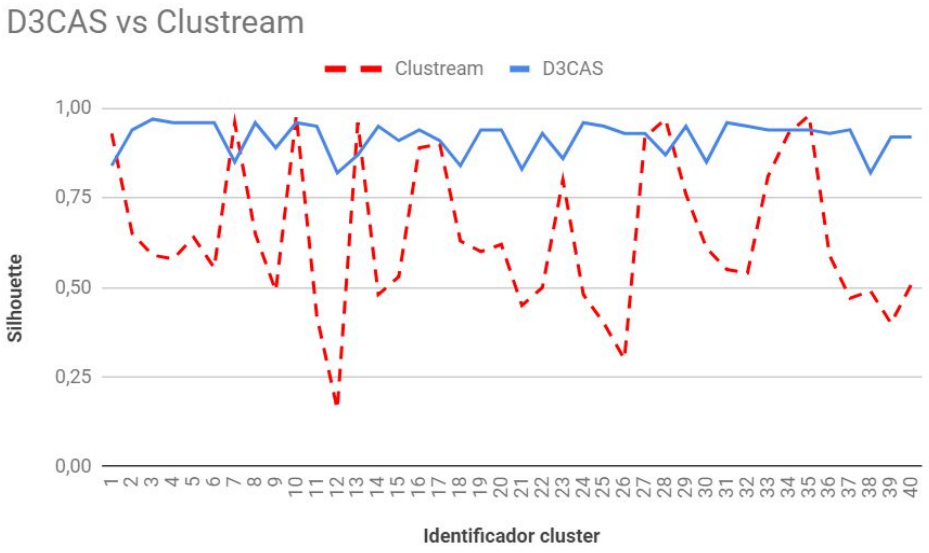


Figura 5.20: Comparación de Silhouette entre Clustream y D3CAS utilizando el dataset de 40 clusters.

Como se puede ver en las Figura 5.17 y Figura 5.19 al realizar pruebas con una cantidad de cluster mayor a diez se puede observar que el algoritmo de Clustream al dar valores promedios de Silhouette de 0.60 está indicando que no puede detectar correctamente todos los clusters del flujo, lo cual, se puede apreciar en las Figura 5.17 y Figura 5.19, donde el algoritmo de Clustream está agrupando varios clusters en uno solo cluster y en otros casos está detectando varios cluster donde solamente hay uno.

En cambio, D3CAS, está dando valores promedio de Silhouette mayores a 0.92, lo que indica que este está detectando correctamente todos los clusters, esto se puede verificar al observar las Figura 5.17 y Figura 5.19. También se puede observar la comparación directa del índice de Silhouette clusters por clusters entre Clustream y D3CAS en las Figura 5.18 y Figura 5.20.

No solo se realizaron comparaciones con datasets 10K21C y 10K44C (de 21 y 44 clusters respectivamente) sino que también con los datasets 10K13C, 10K17C, 10K21C, 10K30C, 10K40C, 10K50C, 10K60C, 10K70C y 10K80C y los resultados fueron los siguientes:

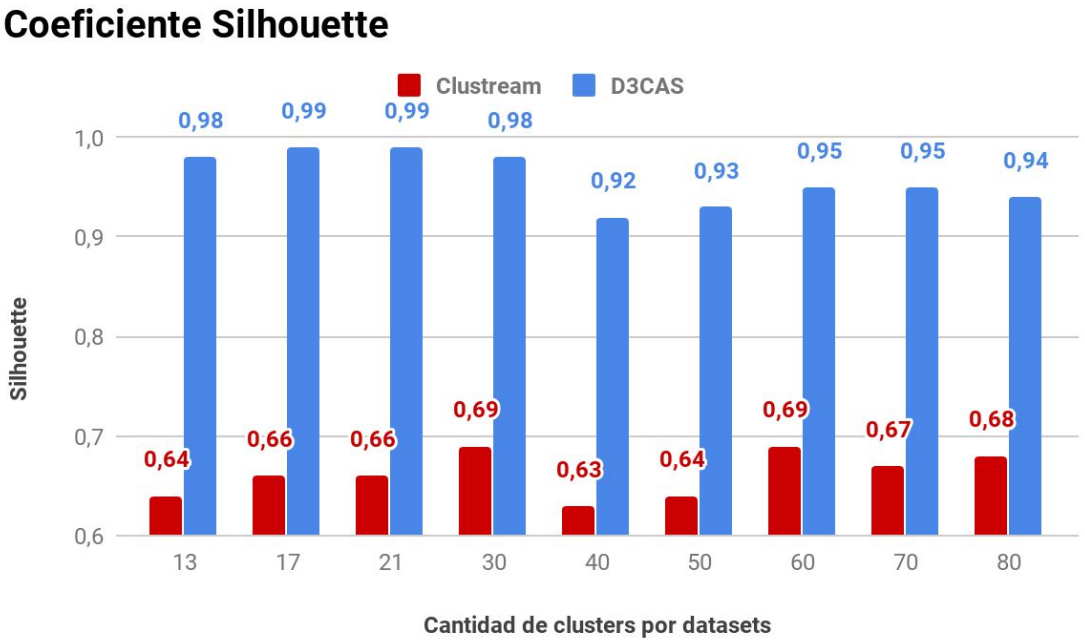


Figura 5.21: Comparación de coeficiente de Silhouette para dataset con 13,17,21,30,40,50,60,70,80 clusters

La Figura 5.21 muestra la comparación de los dos algoritmos con los datasets mencionados anteriormente y a partir de ella se puede analizar que cuando hay una cantidad de clusters mayor a 10 podemos observar que D3CAS da mejores resultados con respecto a Clustream, como se ve en la Figura 5.21, para todos los datasets comparados clustream da un valor de Silhouette promedio de 0.65 aproximadamente mientras que D3CAS da un Silhouette

promedio de 0.95 aproximadamente, por lo tanto, se puede apreciar que en estos casos se observa una diferencia de 0,30 a favor de D3CAS y además se puede concluir que D3CAS detecta correctamente los clusters de un dataset sin verse afectado por la cantidad de clusters que haya en el flujo/dataset (mientras que Clustream si se ve afectado) y genera un modelo (0.95 - 0.65) / 0.65 x 100 = 46 % mejor que Clustream.

Comparación con clusters con formas arbitrarias (no-esféricos)

Hasta ahora se han realizado pruebas con clusters que tienen formas esféricas y que se encuentra bien marcada su separación, en esta sección se propone realizar una comparación de ambos algoritmos utilizando flujos de datos que contenga clusters con formas arbitrarias y aleatorias ya que hay muchas aplicaciones donde la distribución de los datos no solamente contienen formas esféricas sino que también presentan clusters con formas irregulares, como por ejemplo se puede encontrar flujos de datos de monitoreo de tráficos de redes [14] [35].

Para estas pruebas, se tomó la decisión de utilizar datasets que son conocidos para este tipos de análisis o benchmarks, entre estos, se van a utilizar los datasets que ofrece el kit de herramientas “CLUTO” [53], desarrollado por el departamento de Computer Science de la universidad de Minnesota.

Los datasets de CLUTO que se van a utilizar son los siguientes:

- **cluto-t4-8k:** Contiene 8.000 datos, que se encuentra distribuidos entre 6 clusters con formas irregulares/arbitrarias y algunos de los datos representan ruido o noise (datos que no pertenecen a ningún grupo).
- **cluto-t8-8k:** Contiene 8.000 datos, que se encuentra distribuidos entre 8 clusters con formas irregulares/arbitrarias y algunos de los datos representan ruido o noise (datos que no pertenecen a ningún grupo).
- **cluto-t7-10k:** Contiene 10.000 datos, que se encuentra distribuidos entre 9 clusters con formas irregulares/arbitrarias y algunos de los datos representan ruido o noise (datos que no pertenecen a ningún grupo).

También se va a usar un dataset conocido como **diamond9**[54], el cual está compuesto por 3000 datos distribuidos entre 9 clusters con forma de diamante, en este datasets no hay datos que representen ruido.

A continuación se ilustran los datasets originales sin aplicar sin aplicar ninguna técnica sobre los datos que contienen:

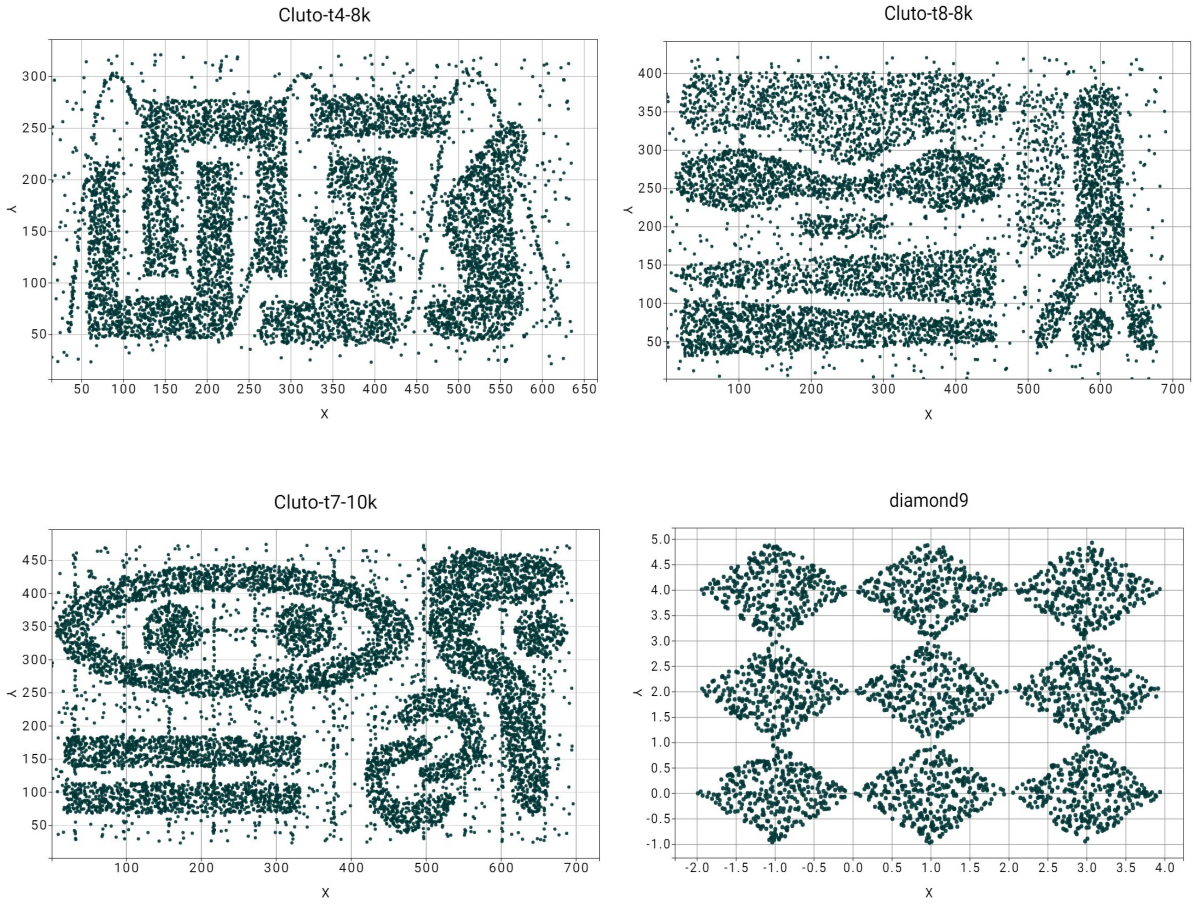


Figura 5.22: Ilustración de datasets de formas arbitrarias

Para medir la validez en este tipo de pruebas no se puede utilizar Silhouette ya que los clusters no son esféricos y no hay una clara separación entre ellos, por lo que las características de compacidad y separación que utiliza Silhouette no son aplicables, por ende para estos casos se tiene que utilizar un método de evaluación de tipo externo. Dicho esto, para estas pruebas se decide utilizar el índice de **pureza de un clusters** el cual está definida de la siguiente forma:

$$pur = \frac{\sum_{i=1}^{K} \frac{|C_i^d|}{|C_i|}}{K} \times 100\%,$$

donde K es el número total de clusters, $\left|C_i^d\right|$ representa la cantidad de elementos agrupados en el cluster i por el algoritmo de clustering y $\left|C_i\right|$ representa la cantidad total de elementos etiquetados para el cluster i. Por lo tanto, intuitivamente, la pureza mide la cantidad de elementos que fueron correctamente agrupados según la etiqueta (label) que representa el cluster real en el dataset.

Podemos aplicar este método, ya que lo datasets de CLUTO, proveen para cada datos el label o etiqueta que representa el cluster a donde está asignado dicho elemento. A continuación se muestran los resultados obtenidos:

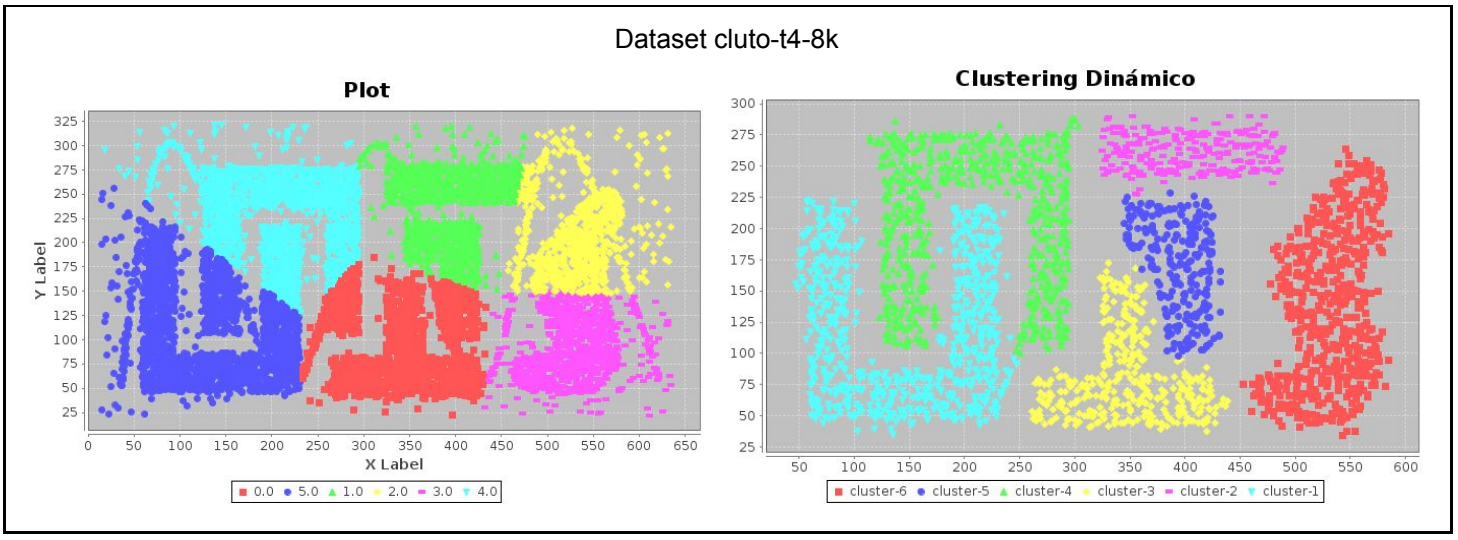


Figura 5.23: a la izquierda resultados de Clustream con pureza = 47% y a la derecha resultados de D3CAS con pureza = 90%

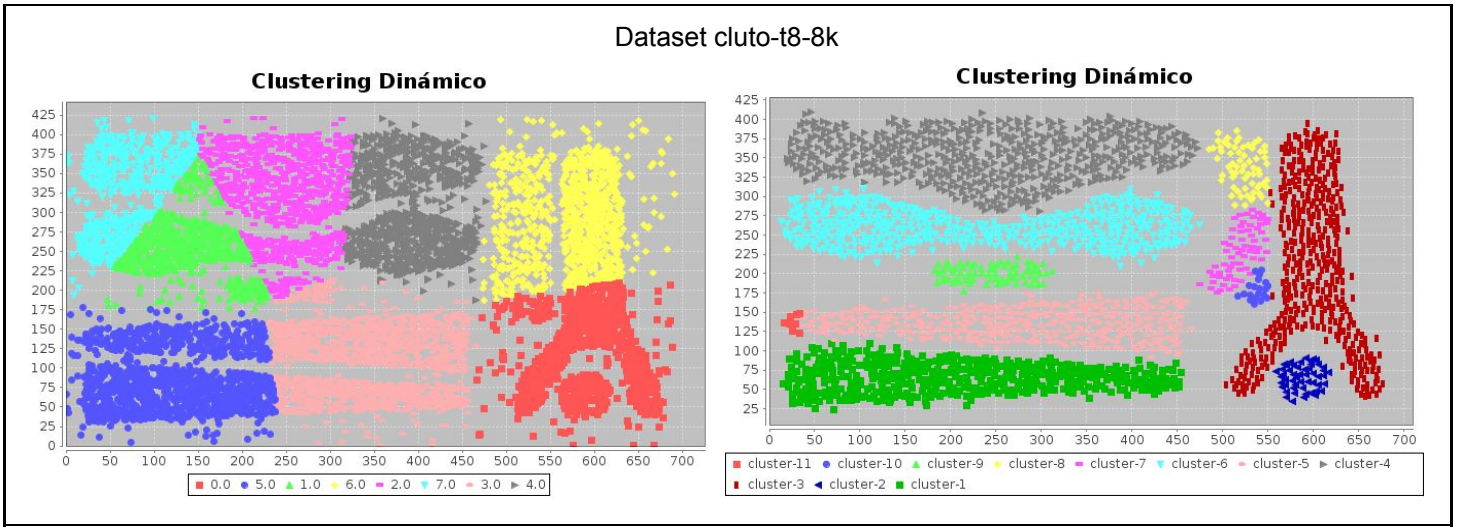


Figura 5.24: a la izquierda resultados de Clustream con pureza = 39% y a la derecha resultados de D3CAS con pureza = 93%

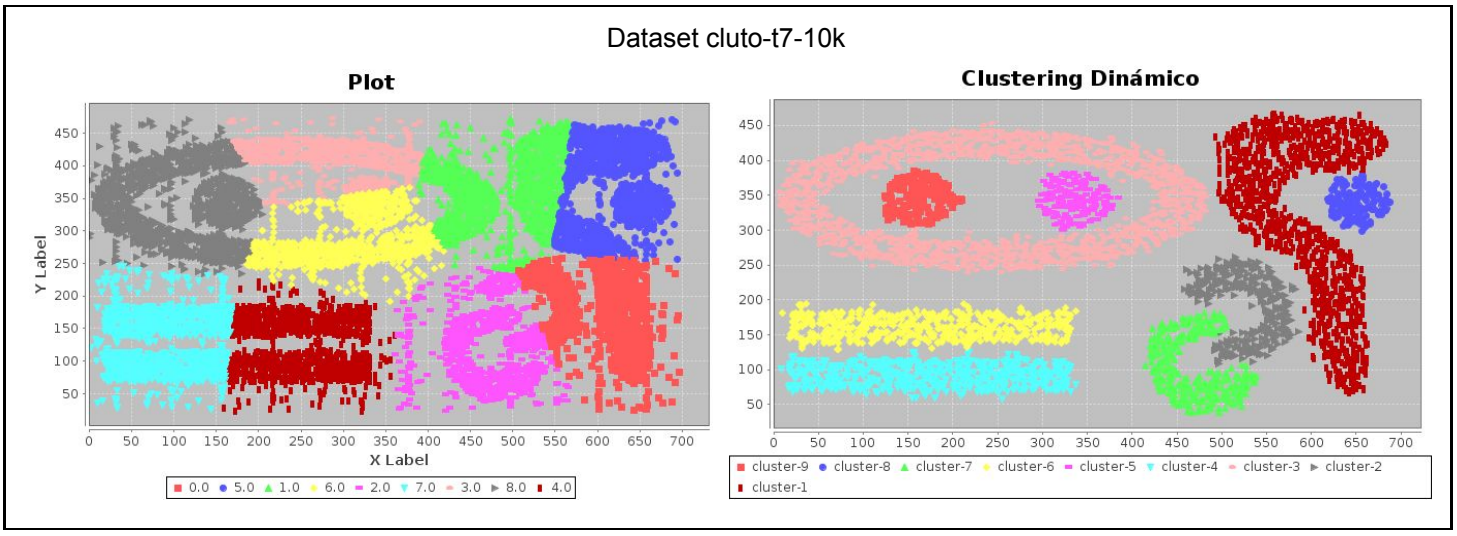


Figura 5.25: a la izquierda resultados de Clustream con pureza = 27% y a la derecha resultados de D3CAS con pureza = 92%

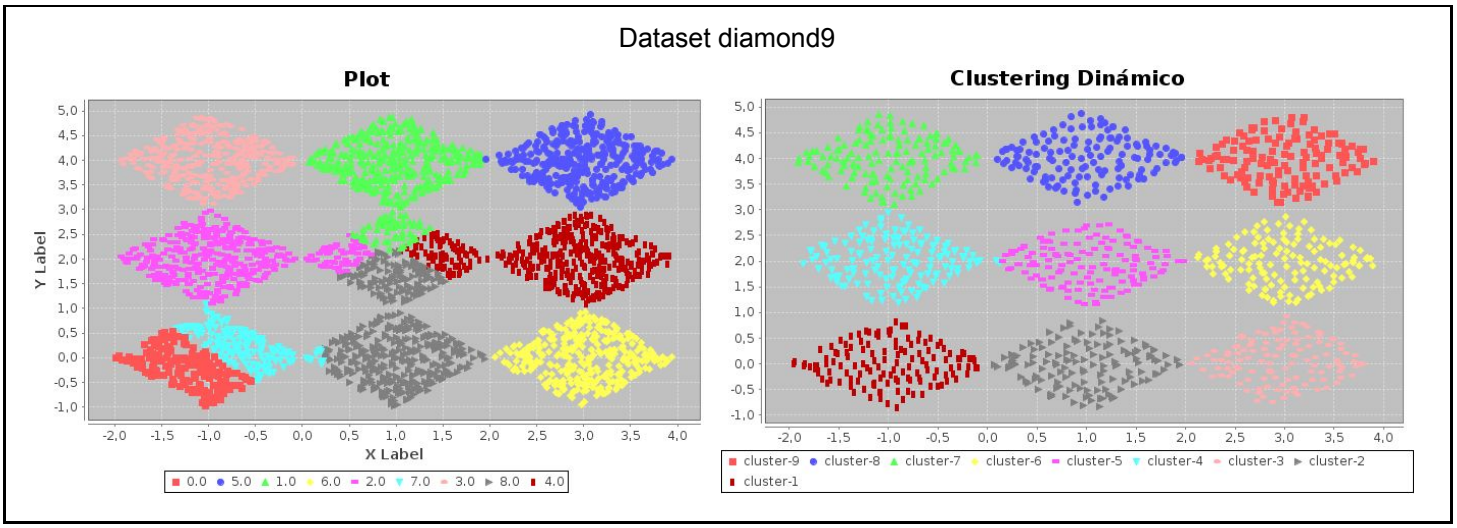


Figura 5.26: a la izquierda resultados de Clustream con pureza = 72% y a la derecha resultados de D3CAS con pureza = 100%

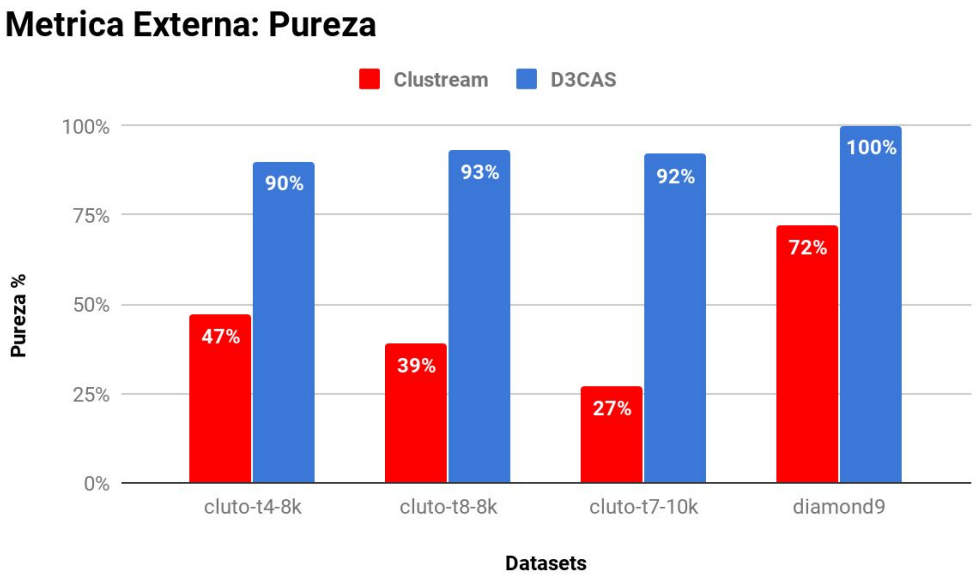


Figura 5.27: Comparación de pureza Cluto entre Clustream y D3CAS.

Como se puede ver en las Figuras 5.{23, 24, 25, 26}, en este tipo de flujos de datos, D3CAS detecta correctamente los grupos que se encuentra entre los datos (salvo en el flujo de cluto-t8-8k, que uno de los grupos fue separado en varios), mientras que en el caso de Clustream no pudo detectar correctamente ninguno de los grupos. Todo esto queda reflejado en los resultados de pureza obtenidos para los dos algoritmos que se puede apreciar gráficamente en la Figura 5.27. Por lo tanto, en este tipo de pruebas podemos afirmar que D3CAS es capaz de detectar clusters con formas arbitrarias/irregulares y además obtiene mejores resultados que Clustream.

Otra característica de suma importancia para destacar es que la implementación de D3CAS puede detectar y filtrar los datos que representan ruido, esto se puede apreciar fácilmente en las Figuras 5.{23, 24, 25, 26}, como se puede ver en estas imágenes no se muestra ningún punto que se encuentre alejado de los clusters formados o entre los clusters. Esta característica no es satisfecha por Clustream, la cual, como se ven en las Figuras 5.{23, 24, 25, 26}, toma los datos que representan ruidos como parte de los clusters.

El motivo por el cual Clustream no puede detectar formas arbitrarias/irregulares, ni tampoco detectar los datos que representan ruido, se debe a que Clustream utiliza un algoritmo de tipo de partición para la detección de los clusters, específicamente, el algoritmo de Lloyd[57], comúnmente llamado Kmeans, y la particularidad de este tipo de algoritmos es que sólo forman clusters con formas esféricas. Mientras que D3CAS, utiliza un algoritmo de densidad, específicamente el algoritmo DBSCAN[35], donde justamente este tipos de algoritmos

permiten detectar grupos con formas irregulares y además detectar los datos que representan ruido.

Reducción del Flujo de datos

En la implementación de D3CAS, la etapa online es la encargada de procesar los datos que llegan al flujo y generar los micro-clusters que se utilizaran en la etapa offline. La cantidad de estos micro-clusters depende del parámetro e, el cual determina el radio del micro-cluster que representa a los datos originales que se encuentran dentro de su área, por lo tanto, a mayor radio, menor cantidad de micro-clusters que representan al flujo.

Como último experimento se propuso evaluar, para un mismo dataset, cual es la calidad lograda dependiendo el radio del micro-clusters, lo cual también se puede analizar como la calidad que se obtiene dependiendo la cantidad de micro-clusters generados.

Para esta prueba se utiliza un flujo de datos que utiliza el dataset 100K14C (el mismo que se utilizó para la prueba de detección dinámica) con el objetivo de consumir el flujo con distintos parámetros de e a fin de generar distintos cantidades de micro-clusters y analizar cuánta variación en los resultados se puede encontrar.

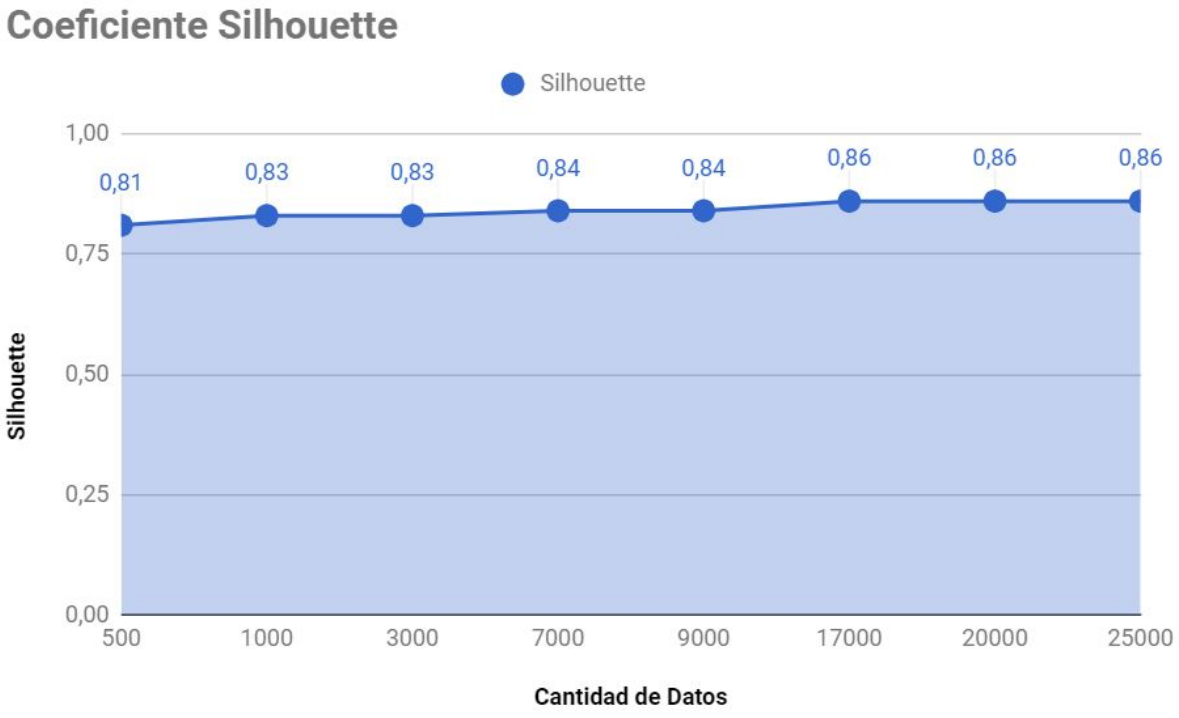


Figura 5.28: Coeficientes de Silhouette dependiendo la cantidad de micro-clusters

Como se ve en la Figura 5.28, utilizando 500 micro-clusters para representar todo el conjunto de datos se obtiene un coeficientes de Silhouette del 0.81, lo cual prueba que se está detectando correctamente los distintos grupos y además indica que con 500 micro-clusters que utiliza 0,5% del espacio que consumen todos los datos se puede representar correctamente las características que tiene el flujo. Además, con 25.000 datos, que utiliza un 25% del espacio del flujo, se observa un coeficientes de Silhouette del 0.86, marcando una diferencia de 0.05 con respecto a la prueba anterior, esto indica que al tener una representación más específica por tener más micro-clusters no genera tanta diferencia entre los resultados obtenidos, ya que la mejora sólo es del 5% ((0.86 - 0.81) / 0.05 x 100 = 5%).

Por último, siempre conviene mantener la mínima cantidad de micro-clusters para reducir el tiempo de ejecución de la etapa offline ya que al ser la encargada de generar los modelos tiene que iterar sobre los micro-clusters generados repetidas veces.

Conclusión

En primer lugar, en esta tesina se estudió y presentó al lector, los conocimientos esenciales para comprender las características, problemáticas y tratamiento de los flujos de datos, a fin de ponerlo en contexto en torno al tema central de esta tesina, la cual centra su foco en el estudio y análisis de las técnicas de clustering dinámicas sobre flujos de datos.

Luego, por medio del estudio y análisis presentado sobre los algoritmos del estado del arte (state-of-the-art) para la detección de clusters en flujos de datos, se brindó al lector todas las consideraciones, requerimientos y características necesarias que tiene que tener un algoritmo ideal de clustering, destacando y llegando a la deducción que es imposible reunir todas las ventajas mencionadas en un solo algoritmo debido a la restricciones naturales que imponen los flujos de datos. A parte de esto, por medio de las ventajas y limitaciones presentada para cada algoritmo estudiado se le deja al lector los conocimientos necesarios para que pueda elegir la metodología de procesamiento que más se ajuste a su problema o caso de estudio.

Todos los algoritmos analizados afirman que son escalables, pero las experimentaciones de estas técnicas fueron llevadas a cabo sobre un enfoque no distribuido y sin paralelismo, es decir, sobre una sola máquina, lo que hizo que esta tesina fuese más desafiante ya que la implementación de cualquiera de los algoritmos investigados requería comprender no solo el algoritmo sino también tener en cuenta los siguientes factores: entender sobre qué arquitectura y modelo fueron diseñados, tener en cuenta las restricciones que imponen los flujos de datos y adaptar la lógica de dichas técnicas a un entorno distribuido, específicamente, sobre el modelo de procesamiento de Spark. En este punto, podemos llegar a la conclusión de que hay bastante investigación y estudio para las técnicas de clustering sobre flujos de datos pero no centrados en los campos de la programación distribuida.

Al utilizar el framework Spark Streaming, cuya arquitectura está basada en los modelos de MapReduce y DryadLINQ, introdujo otros desafíos adicionales ya que algunas de las características necesarias para implementar técnicas de clustering no están disponibles o no son triviales en este modelo, lo cual llevó a probar distintas soluciones para resolver estos desafíos.

Por ejemplo, uno de los desafíos más importantes fue el alcance o visibilidad global de los datos que se necesitan sobre las etapas del proceso de clustering. Todas las metodologías

de procesamiento empleadas sobre las técnicas estudiadas asumen un alcance global de todos los datos que llegan al flujo en una ventana de tiempo, en cambio, en la arquitectura de Spark no es posible, pues cada nodo Worker solo conoce su porción de datos a procesar.

Pese a todos los desafíos presentados, el objetivo de diseño e implementación de una técnica de clustering dinámica sobre un flujo de datos se pudo lograr satisfactoriamente, dando origen al algoritmo D3CAS, el cual brinda una solución capaz de trabajar en un entorno distribuido y escalable gracias a que la implementación fue llevada sobre el motor de Spark Streaming. Además, por las característica de esta técnica, tanto por ser dinámica y estar basada en un algoritmo de densidad (DBSCAN), se la puede considerar como la primer técnica de clustering implementada sobre Spark con estas características debido a que a la fecha no hay publicación de algún trabajo similar.

Los resultados obtenidos en los experimentos y comparaciones de D3CAS junto a otras técnicas sobre Spark fueron alentadores ya que se obtuvieron grupos de buena calidad tanto para grupos esféricos como para grupos de formas arbitrarias. Obviamente, hay margen de mejora y se pueden realizar optimizaciones en el sistema para mejorar los resultados de la agrupación. Además, pese a estar trabajando con un motor que permite procesar de forma distribuida e implementar la técnica bajo esta arquitectura, las pruebas realizadas fueron llevadas en un entorno distribuido simulado sobre un entorno local de una sola máquina, ya que no se obtuvo a disposición un cluster de máquinas configurado con Spark para llevar a cabo las pruebas. Esta tesis, por lo tanto, brindó una prueba de concepto para la implementación de un algoritmo de streaming clustering en un entorno escalable y distribuido, siendo este tipo de técnica no del todo popular (tendencia) en las corrientes principales a pesar de tener muchos casos de uso útiles.

Trabajos Futuros

Como trabajos futuros se proponen las siguientes investigaciones y pruebas que permitan medir y mejorar el rendimiento del algoritmo presentado:

Como primer medida, se propone el estudio y análisis para la configuración de un cluster real de computadoras con el framework Apache Spark Streaming para llevar a cabo pruebas del algoritmo sobre un entorno real y consumiendo un flujo de datos real, como por ejemplo, se podría consumir los flujos de datos que brindan los servicios Twitter mediante su API.

También se proponen como trabajo futuro, continuar con la investigación sobre el framework de Spark a fin de encontrar nuevas características, prestaciones o optimizaciones que permitan mejorar el rendimiento del algoritmo implementado. Específicamente se propone llevar a cabo una investigación sobre los métodos de comunicación de datos entre los nodos de Spark a fin de llevar a cabo pruebas de conceptos sobre estos métodos y realizar comparaciones tanto a nivel de resultados como también a nivel de consumo de recursos como memoria, procesamiento, overhead de comunicación, consumo energético, etc.

Relacionado con el punto anterior, se propone como un desafío mayor, el estudio y análisis de la posibilidad de implementar técnicas de clustering bajo el modelo de programación en GPUs, la cual es un área donde no hay mucha investigación sobre este tipo de técnicas.

Referencias bibliográficas

1. Aggarwal C.C. (2009) **Data Streams: An Overview and Scientific Applications**. In: Gaber M. (eds) Scientific Data Mining and Knowledge Discovery. Springer, Berlin, Heidelberg.

2. **Models and Issues in Data Stream Systems**: Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom, Department of Computer Science, Stanford University Stanford, CA 94305.

3. **Issues in Data Stream Management** - Lukasz Golab and M. Tamer Ozsu, University of Waterloo, Canada.

4. Aggarwal 2007 - **Data Streams, Models and Algorithms.**

5. Charu C Aggarwal, Jiawei Han, Jianyong Wang, and Philip S Yu. **A framework for clustering evolving data streams.** In Proceedings of the 29th international conference on Very large data bases-Volume 29, pages 81–92. VLDB Endowment, 2003.

6. **A survey on data stream clustering and classification**, Hai-Long Nguyen, Yew-Kwong Woon, Wee-Keong Ng.

7. **Understanding Concept Drift** - Geoffrey I. Webb, Loong Kuan Lee, Bart Goethals, Franc¸ois Petitjean

8. Gama J, Rodrigues P (2009) **An overview on mining data streams**, vol 206 of Studies in computational intelligence, pp 29–45. Springer, Berlin

9. Wang H, Yu PS, Han J (2005) **Mining data streams**. Springer, US, pp 777–792

10. D. Terry, D. Goldberg, D. Nichols, and B. Oki. **Continuous queries over append-only databases**. In Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data, pages 321–330, June 1992.

11. Domingos P, Hulten G (2000) **Mining high-speed data streams**. In: Proceedings of the sixth ACM SIGKDD international conference on knowledge discovery and data mining, pp 71–80, 347107. ACM 36. Dork M, Gruen D, Williamson C, Carpendale S (2010) A visual backchannel for large-scale events.

12. Hulten G, Spencer L, Domingos P (2001) **Mining time-changing data streams**. In: Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining, pp 97–106, 502529.

13. Zhang P, Zhu X, Shi Y, Wu X (2009) **An aggregate ensemble for mining concept drifting data streams with noise**, vol 5476 of Lecture notes in computer science, pp 1021–1029. Springer, Berlin.

14. Cao F, Ester M, Qian W, Zhou A. **Density-based clustering over an evolving data stream with noise**. In: Proceedings of the 2006 SIAM international conference on data mining, pp 328–339.

15. Alexander Gepperth, Barbara Hammer. **Incremental learning algorithms and applications**. European Symposium on Artificial Neural Networks (ESANN), 2016, Bruges, Belgium.

16. Street WN, Kim Y (2001) **A streaming ensemble algorithm (sea) for large-scale classification**. In: Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining, pp 377–382.

17. Zeng H-J, He Q-C, Chen Z, Ma W-Y, Ma J (2004) **Learning to cluster web search results**. In: Proceedings of the 27th annual international ACM SIGIR conference on research and development in information retrieval, pp 210–217, 1009030.

18. Sow D, Biem A, Blount M, Ebling M, Verscheure O (2010) **Body sensor data processing using stream computing**. In: Proceedings of the international conference on multimedia information retrieval, pp 449–458, 1743465.

19. Aggarwal CC, Yu P (2005) **Online analysis of community evolution in data streams.** In: Proceedings of the SIAM international conference on data mining, pp 56–67.

20. Characterizing the 2016 U.S. Presidential Campaign using Twitter Data (2016). Ignasi Vegas, Tina Tian Department of Computer Science Manhattan College New York, USA. Wei Xiong Department of Information Systems Iona College New Rochelle, USA. - (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 7, No. 10, 2016.

21. Data Mining for Social Media Analysis: Using Twitter to Predict the 2016 US Presidential Election (2016). Kabir Ismail Umar Department of Information Technology, ModibboAdama University of Technology Yolakabir.ismail@mautech.edu.ng - Fatima Chiroma Software Development Department, American University of Nigeria Fatima.chiroma@aun.edu.ng.

22. **RT^2M: Real-Time Twitter Trend Mining System (2013)**. Min Song ; Meen Chul Kim - DOI: 10.1109/SOCIETY.2013.19.

23. **Social Media Data Mining: A Social Network Analysis Of Tweets During The 2010-2011 Australian Floods.** (2011) France Cheong, RMIT University, france.cheong@rmit.edu.au - Christopher Cheong RMIT University, christopher.cheong@rmit.edu.au.

24. **Data Mining for Network Intrusion Detection**: How to Get Started Eric Bloedorn, Alan D. Christiansen, William Hill, Clement Skorupka, Lisa M. Talbot, Jonathan Tivel.

25. Lee, W., and S. Stolfo [1998]. **"Data Mining Approaches for Intrusion Detection"**, in Proceedings of the 7th USENIX Security Symposium, San Antonio, TX.

26. Clifton, C., and G. Gengo [2000]. **"Developing Custom Intrusion Detection Filters Using Data Mining"**, 2000 Military Communications International, Los Angeles, California, October 22-25.

27. **Mining Frequent Patterns in Data Streams at Multiple Time Granularities** - Chris Giannella, Jiawei Hany, Jian Peiz, Xifeng Yany, Philip S. Yu. Indiana University.

28. J. Dean and S. Ghemawat. **MapReduce: Simplified data processing on large clusters.** In OSDI, 2004.

29. **DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.** Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, Jon Currey. Microsoft Research Silicon Valley joint affiliation, Reykjavík University, Iceland.

30. **Learning Spark, Lightning-Fast Big Data Analysis** - By Matei Zaharia, Holden Karau, Andy Konwinski, Patrick Wendell.

31. **High Performance Spark, Best Practices for Scaling and Optimizing Apache Spark** - By Holden Karau, Rachel Warren. June 2017: First Edition.

32. Documentación oficial de **Apache Spark** - https://spark.apache.org/docs

33. **Spark: Cluster Computing with Working Sets.** By Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica University of California, Berkeley.

34. **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.** By Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur

Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica University of California, Berkeley.

35. **A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise.** by Martin Ester , Hans-Peter Kriegel , Jörg Sander , Xiaowei Xu.

36. **Ackermann MR, Märtens M, Raupach C, Swierkot K, Lammersen C, Sohler C. StreamKM++: A clustering algorithm for data streams.** ACM J Exp Algorithmics. 2012; 17(1):173–187.

37. **k-means++: The Advantages of Careful Seeding**, David Arthur and Sergei Vassilvitskii.

38. Zhang X, Furtlehner C, Sebag M. **Data streaming with affinity propagation**. In: ECML/PKDD (2). Berlin: Springer Berlin Heidelberg: 2008. p. 628–43.

39. Tian Zhang, **Data Clustering for Very Large Datasets Plus Applications**, Computer Sciences Dept. at Univ. of Wisconsin-Madison, 1996.

40. Clark F. Olson, **Parallel Algorithms for Hierarchical Clustering**, Technical Report, Computer Science Division, Univ. of California at Berkeley, Dec.,1993.

41. Guttman A (1984) **R-trees: a dynamic index structure for spatial searching.** SIGMOD, Boston, pp 47–57.

42. Seidl T, Assent I, Kranen P, Krieger R, Herrmann J (2009). **Indexing density models for incremental learning and anytime classification on data streams**.

43. Philipp Kranen · Ira Assent · Corinna Baldauf · Thomas Seidl. **The ClusTree: indexing micro-clusters for anytime stream mining.**

44. Aggarwal CC, Han J, Wang J, Yu PS (2004) **A framework for projected clustering of high dimensional data streams.** VLDB, Toronto, pp 852–863.

45. **BIRCH: A New Data Clustering Algorithm and Its Applications** - TIAN ZHANG, RAGHU RAMAKRISHNAN, MIRON LIVNY. Computer Sciences Department, University of Wisconsin, Madison, WI 53706,U.S.A.

46. **Effectiveness of the Euclidean distance in high dimensional spaces (2015)** - Shuyin Xia, Zhongyang Xiong, Yueguo Luo, WeiXu, Guanghua Zhang.

47. On Clustering Validation Techniques. MARIA HALKIDI, YANNIS BATISTAKIS, MICHALIS VAZIRGIANNIS. Department of Informatics, Athens University of Economics & Business, Patision 76, 10434, Athens, Greece (Hellas).

48. Theodoridis, S. and Koutroubas, K. **(1999). Pattern Recognition**. Academic Press.

49. Berry, M.J.A. and Linoff, G. **(1996). Data Mining Techniques For Marketing, Sales and Customer Support.** John Wiley & Sons, Inc., USA.

50. Evaluating and Analyzing Clusters in Data Mining using Different Algorithms - N. Sunil Chowdary , D. Sri Lakshmi Prasanna , P. Sudhakar. Department of CSE, Sri Sarathi Institute of Engineering and Technology, Nuzvid,-521201, Krishna District, A.P - ISSN 2320–088X.

51. http://www.noahlab.com.hk/

52. https://github.com/huawei-noah/streamDM

53. http://www.cs.umn.edu/~cluto

54. Salvador, S. and Chan, P.,**Determining the Number of Clusters/Segments in Hierarchical clustering/Segmentation Algorithm,** ICTAI 2004,576-584.

55. https://github.com/FakenMC/generateData

56. Fachada, N., Figueiredo, M.A.T., Lopes, V.V., Martins, R.C., Rosa, A.C.,**Spectrometric differentiation of yeast strains using minimum volume increase and minimum direction change clustering criteria**, Pattern Recognition Letters, vol. 45, pp. 55-61 (2014),doi:http://dx.doi.org/10.1016/j.patrec.2014.03.008.

57. Lloyd, Stuart P. (1982), **"Least squares quantization in PCM"**, IEEE Transactions on Information Theory, 28 (2): 129–137